Fall 12-2023

Static Binary Rewriting for ROP Gadget Removal

Hans Verhoeven

DAKOTA STATE UNIVERSITY

STATIC BINARY REWRITING FOR ROP GADGET REMOVAL

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

Doctor of Philosophy

in

Computer Science

December, 2023

By
Hans Verhoeven

Dissertation Committee:

Michael Ham
Kyle Murbach
Yong Wang

DISSERTATION APPROVAL FORM

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: Hans Verhoeven

Dissertation Title:
STATIC BINARY REWRITING FOR ROP GADGET REMOVAL

Graduate Office Verification: DocuSigned by: Brianna Mae Feldkamp F44C3D9EE21C417... Date: 12/12/2023

Dissertation Chair/Co-Chair: DocuSigned by: Michael Ham 499B29D70C7542A... Date: 12/12/2023
Print Name: Michael Ham

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

Committee Member: DocuSigned by: Kyle Murbach 8B9480BB16420... Date: 12/12/2023
Print Name: Kyle Murbach

Committee Member: DocuSigned by: Dr. Yong Wang 7A6B505BC7B645E... Date: 12/12/2023
Print Name: Dr. Yong Wang

Committee Member: _____ Date: _____
Print Name: _____

Committee Member: _____ Date: _____
Print Name: _____

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

ACKNOWLEDGMENTS

With this major milestone in my life being completed, I would like to acknowledge everyone that has helped me in completing it. The first group of individuals I want to acknowledge are my committee members. I want to thank them for their guidance throughout this process and the expertise they provided me with. I also greatly appreciate the time they sacrificed to help me. Their insights helped me create the best work that I could make. I would also like to thank the faculty here at Dakota State University. I came to DSU with no prior knowledge in Computer Science, but they taught me almost everything I know today. They also had to deal with my barrage of never-ending questions.

I also would not be here without the support of my family and friends. My mom dedicated almost 20 years of her life to homeschooling me and my siblings. My dad provided for us and set aside time to help me understand Computer Science topics in college. Both of them also supported me immensely in everything I did and I cannot thank them enough. My siblings and friends provided me with encouragement and entertainment. Without their support, I do not know if I would have had the drive to get through this endeavor. So, I am truly grateful for them.

ABSTRACT

Return-Oriented Programming (ROP) is an exploitation technique that is commonly used by malicious users. It works by leveraging return statements in binaries to gain control over the execution of programs. Some mitigations for ROP include changing the binary during compilation time, rewriting the binary after compilation, and adding runtime checks to the binary. The focus of this study was rewriting the binary after compilation. Rewriting during compilation time requires end users to have access to source code, which, in most cases, they will not. Adding runtime checks adds additional overhead to the target binary.

The areas this study aimed to improve in the binary rewriting space were twofold. The first was improving static binary rewriting. This was done by attempting to see if the amount of information needed to correctly rewrite a binary could be reduced compared to other tools. The second area was attempting to use static binary rewriting to reduce the number of potential ROP gadgets in a binary. The ROP gadgets that were targeted were those created by splitting an instruction that contains a return in them to create new ROP gadgets. This was chosen because most current tools focused on the safety of standard returns from function ends.

To determine if static binary rewriting could be used to reduce the amount of ROP gadgets created from mid-instruction ROP gadgets, a design science approach was taken. There were two artifacts that were created through two design cycles. The first artifact aimed to create a static binary rewriter that collected minimal amount of information from binaries. The second artifact built upon the first artifact and attempted to use it to remove instructions that contained a mid-instruction return. After the removal of the mid-instruction return, the

second artifact inserted instructions that allowed for the same functionality of the binary, but without the return byte.

Keywords: reassemblable disassembler, trampoline rewriter, return-oriented programming, ROP, ROP gadgets, static binary rewriting

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,

Hans Verhoeven

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Binary rewriting is taking a compiled binary and modifying the instructions it executes during runtime without having to recompile it. Recently, a large number of binary rewriters/patchers have been released (Schulte et al., 2022; Wang et al., 2015; Wenzl et al., 2019). These binary rewriters fall into two main categories: static and dynamic. The dynamic rewriters require the binary to be running to perform modifications and the static rewriters can make their modifications without needing the binary to be running. Binary rewriting is also being used to add or enhance security features in binaries (CHENG et al., 2014; Onarlioglu et al., 2010; Xu et al., 2020). The main security feature this study is going to investigate is Return-Oriented Programming (ROP) defense.

The focus of this study was to determine if it was possible to create a static binary rewriter that can eliminate ROP gadgets from binaries and create a list of instructions to eliminate each specific ROP gadget. ROP attacks are still prevalent today and being able to hinder them would aid the security of any programs that a user would run. In addition, a list of instructions used for the elimination of ROP gadgets could be shared and used in other binary rewriters.

Background of the Study

Almost all aspects of our lives now interact with some form of application/software. This massive integration of technology into people's lives means that the security of the devices, in turn, is also security for people themselves. These protections can come in two

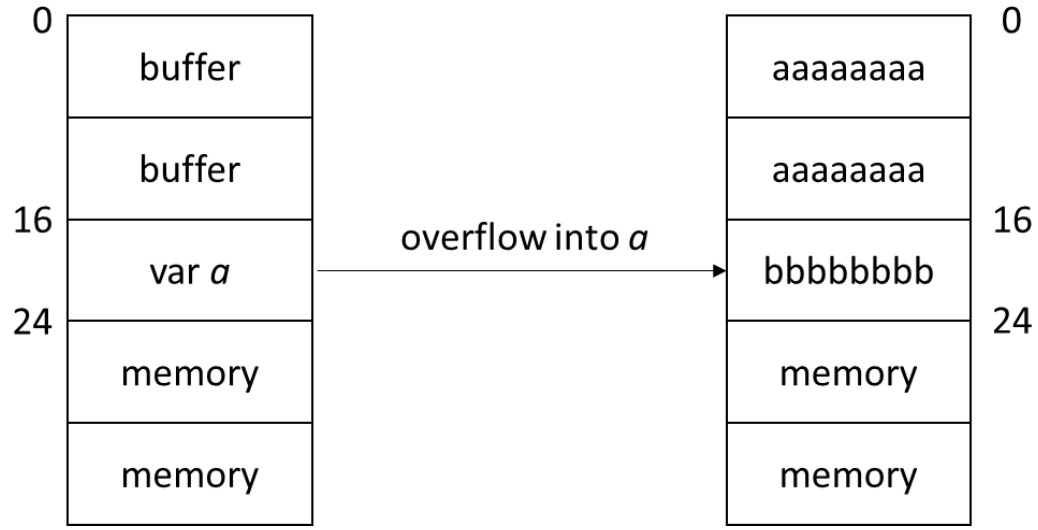
main forms. The first form is software security (McGraw, 2004), which aims to add the protections to the software itself. The software protections attempt to secure data variables in the program or add checks during execution to ensure program integrity. Another method attempts to alleviate the problem not on the developer's side, but on the hardware side of the device (Rostami et al., 2014). Enforcing the protections at a hardware level can help hinder attackers since, even if a software vulnerability exists, it would be stopped at the hardware level. Both protections can help protect end users from malicious attacks. These attacks cost the average victim $4,476 dollars on average and 71.1 million users fell victim to these attacks in 2021 (Sharif & Mohammed, 2022). The number of attacks and their severity have also been on the rise (Sharif & Mohammed, 2022).

However, even with these protections, attackers are still constantly trying to find vulnerabilities in the programs that we use. Not every protection is fool-proof and it is simply a matter of time before an exploit is found. Adversaries can leverage a variety of techniques to gain control of systems, leak data from devices, and more. These techniques can lead to actual attacks against the programs and thus harm the users that use the applications. Already this year, Apple had to release patches to two critical security issues that included seven different Apple devices, including iPhones (Gatlan, 2023). Vulnerabilities could allow attackers to steal data from the device or allow attackers to put software or data on the device. One example of this was in 2022 when a campaign was launched against human rights defenders in Mexico using zero-click exploits on iOS versions 15 and 16 (Marczak et al., 2023). Zero-click exploits are dangerous because they take advantage of vulnerabilities in software or apps to infect users without the user doing anything (Fiscutean, 2022). In one example, a messaging app was vulnerable to a zero click and all the attacker had to do was send a message to infect

a user (Fiscutean, 2022). Lastly, Android also had to apply two security patches this March for certain Android phone versions (Arntz, 2023). The next paragraphs discuss some of these types of attacks and review some of the mitigations for them.

One type of attack that attackers can use is a Buffer Overflow attack (Aleph One, n.d.). This type of attack works by taking advantage of improper control of the amount of data that can be written to a buffer. A buffer is a location in computer memory that programs use to store data. The length of a buffer is sometimes statically sized, which means it will be a fixed length. If more data is written to a buffer than the buffer can hold, the memory segments that follow the buffer will begin to be overwritten by this overflow of data. A more specific example is shown in Figure 1. In this example, there is a buffer located at addresses 0 through 15. Additionally, there is a variable with the name *a* found at address 16. If the program attempts to store more than 16 characters into the buffer and no checks stop the user from adding more than 16 characters, the written data will start to "overflow" from the buffer into the variable *a*. To accomplish this, an attacker could take advantage of vulnerable functions, such as *scanf*, that do not properly compare input length to buffer size (CWE, n.d.). An example of filling the buffer up is shown in Figure 1 by writing 16 a's and then putting 8 b's into the variable *a*. This overflow can lead to ROP vulnerabilities in programs. What ROP vulnerabilities are and their impact will be discussed in detail in the following paragraph.

Figure 1 Example buffer overflow



A ROP attack can leverage return statements to gain control of the program (Bhat, 2019). A return statement is an assembly instruction that is mainly found at the end of functions in binaries. Functions are a grouping of instructions that can be invoked. The return instruction instructs the program what address to resume execution at after the function has finished. In some assembly architectures, such as Intel, return instructions can also be created by starting execution at specific offsets in memory to cause a “c3” byte in an instruction to be treated as a return instruction. A ROP attack modifies what address the program resumes execution at after a return instruction is executed. The attacker then chains together small lists of instructions followed by another return instruction. These small lists of instructions and their returns are referred to as ROP gadgets.

An example of how ROP attacks work is shown in Figure 2. First, some function is called in the main program. The function in the example is called *get_input*. Inside of this function is a buffer that stores input and the length of the input is never checked. Since the length is never checked, the function is vulnerable to a buffer overflow. Once the function is

called, this vulnerable buffer will be placed on the stack followed by two values. The stack is a memory structure that programs use to keep track of certain values and the two values are the old RBP value and the return instruction. RBP is a register. Registers hold values that programs use during execution. The RBP value can be ignored because it will get overwritten in this attack and the return instruction is the address that the program will return execution to after the function call ends. Once the return is executed, it will pop/remove the value from the top of the stack and return to that address.

Figure 2 Simple ROP chain

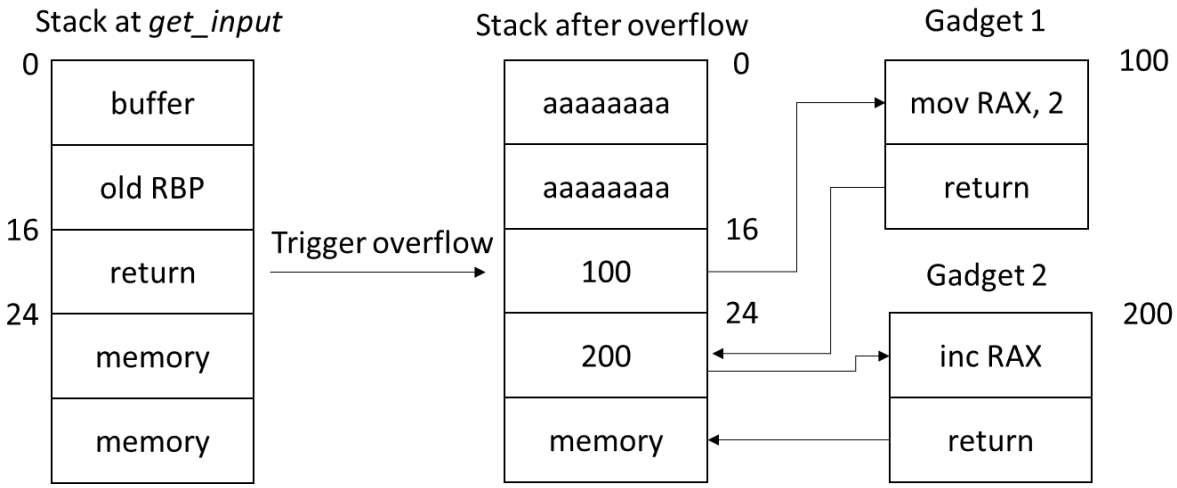


Figure 2 shows what a stack could look like at a given time. Removing an item from the stack allows for a previously added value on the stack to become the top. So, in Figure 2 when the return value is popped, the memory value at 24 will become the top of the stack. This means if multiple values can be written to the stack during the overflow, the program's execution will be controllable. The example in Figure 2 only puts the value 3 into RAX but gives an understanding of how a ROP chain could be built. First, the addresses 100 and 200 are put onto the stack so the next two returns will go to 100 and 200. Next, the first gadget

will put the value 2 into RAX. Then the second gadget will increment the value in RAX making it 3. Finally, execution will return to wherever the next value on the stack is. The next value can be controlled, but this example does not show this.

Some mitigation techniques, such as Address Space Layout Randomization (ASLR) , stack canaries, and Data Execution Prevention (DEP), have been implemented to try to hinder this type of attack (Bierbaumer et al., 2018; Ganz & Peisert, 2017; Marcho, 2019). All these protections can be bypassed by a malicious user. The following paragraphs will describe these protections and how they can be bypassed.

The ASLR mitigation shuffles what location certain instructions are located at during the runtime of a program. The ASLR mitigation can make it more difficult for the attacker to use a buffer exploit to create a ROP gadget chain by shuffling what instructions are at specific addresses. This can be shown by analyzing Figure 2 again. With ALSR enabled in the program, the location of gadget 1 and gadget 2 would not always be at address 100 and 200. This would prevent the attacker from simply writing those two numbers to the stack. The attacker would need to find the location of those gadgets during the execution of the program. An attacker can leverage memory leak vulnerabilities to determine the locations of the gadgets (Planet, 2010). While this prevents hard-coded addresses and makes it more difficult to create working ROP gadget chains, the mitigation is not foolproof (Sporici, 2019).

A stack canary is used to detect buffer overflows. The stack canary mitigation dynamically creates a value when a function is called. The value is saved by the calling routine and is also placed on the stack before the function’s return statement. The purpose of placing the canary value before the return instruction is that if an overflow overwrites the return value it will also overwrite the canary value. Once the return is executed, the canary

value is checked to ensure that it matches the value when the function was called. If the values are not the same, the program will terminate. Memory leaks (Planet, 2010) could leak the value of the canary and this could allow an attacker to make a specifically crafted buffer overflow that would contain the proper value of the canary in it.

DEP defines certain sections of a binary as executable and nonexecutable. The nonexecutable sections are sections that contain bytes related to data and not instructions. Without DEP enabled, these data bytes could be executed leading to additional instructions that an attacker could use. However, ROP exploits target executable instructions. So, while it may limit the number of gadgets that can be used, ROP is still possible with DEP enabled.

Statement of the Problem with Motivation

Binary patching/rewriting takes a compiled binary and modifies it while maintaining program integrity without source code access (Wenzl et al., 2019). Binary rewriting is only able to be used on compiled languages, such as C, because a conversion of source code to machine code is made. Binary rewriting targets this machine code and modifies it without needing to perform the compilation step again.

A wide variety of tools, such as IDA (hex-rays, n.d.), Ghidra (*Ghidra Software Reverse Engineering Framework*, 2019/2023), Radare2 (GitHub, 2012/2023), and Angr (GitHub, 2015/2023) all implement some form of binary patching. IDA, Ghidra, and Radare2 are reverse engineering tools that can disassemble binaries and allow users to insert instructions at chosen locations. Angr is a tool that uses static and dynamic symbol analysis and can use binary patching to insert functionality at specific locations. Using binary rewriting as a method to defend against ROP attacks has been previously researched (Prasad

& Chiueh, 2003; Xu et al., 2020; Xu & Wang, 2022). However, none of these examples target the ROP gadget themselves and add protections around the ROP gadgets to hinder them.

In addition to binary rewriting, two other methods involving compiler-based (Onarlioglu et al., 2010) and runtime-based (CHENG et al., 2014) approaches have been used. However, the compiler-based method either expects the creator of the program to use this method or that the end user has access to source code to compile it themselves. Expecting all software creators to use a compiler-based method when creating their program is overly optimistic. So, putting all users' security in their hands is not a solid approach. Additionally, most vendors will not send source code along with their software for proprietary reasons. This means that most end users will never have access to the source code to compile it themselves. Because of those reasons, compiler-based methods are not optimal for the end user. The runtime-based technique does not modify the binary instructions and thus was not covered in this study.

The goal of this study was to provide access to some of the compiler-based solutions to a binary rewriting technique. This would allow ROP gadgets to be removed from the binary statically and without source code rather than trying to add protections around the ROP gadgets. The lack of needing source code is extremely beneficial for end-users because most programs vendors' releases do not come with the source code. Not having source code means that compiler-based solutions would not be able to be used in many cases. By implementing a binary rewriting approach, end-users can implement the security measures without needing software vendors to release their programs with source code for recompilation. In addition, this newly hardened binary could be integrated with the runtime-based approaches (CHENG

et al., 2014) or, should the static analysis fail, the ROP gadget section could be integrated with existing binary rewriting tools (Schulte et al., 2022).

Purpose of the Study

The purpose of the study was to use a design science research (vom Brocke et al., 2020) (DSR) approach to generate a tool that was able to statically remove ROP gadgets from compiled binaries. Quantitative analysis (Cárdenas, 2019; Sandelowski, 1995) was not chosen because there is a heavy emphasis on comparison of variables. While there were comparisons in the amount of ROP gadgets, the main goal was developing a method of removing them. Qualitative analysis (Sandelowski, 1995) was not chosen due to a lack of open-ended design metrics. Since both quantitative and qualitative approaches did not fit this study, the DSR method to create the tool was chosen.

There were three main reasons for creating a tool that performs these tasks. The first reason was to try to shift ROP protection away from compiler-based approaches. Compiler-based approaches remove the ability of end users to apply the protections themselves. This means that end users must inherently trust developers to apply the security patches. The second reason was to address new methods of static binary rewriting. Most binary rewriters currently attempt to recover basic block information or lift the binary into an intermediate language for label creation. However, having to recover more information from a binary or needing to lift to the intermediate language also allows more room for error. These errors can arise from mislabeling of locations or the inability to recover a control flow graph of the binary. The approach aligned in this research attempted to determine if less information is needed to provide effective static binary patching techniques. Some of the current tools also integrate with additional tools to help either determine control flow or add labels to the lifted

intermediate language. By doing this, more onus is put onto end users to install and maintain additional software to apply the security.

The third reason involved the current implementation of ROP protection tools. Current protections either use compiler-based, return-based, or runtime-based protections. The issues around compiler-based solutions involving the lack of source code for end users or trusting vendors to use the compiler-based solution when compiling were previously mentioned. The return-based rewriters place protections around identified function return locations or other return-based instructions. Issues with this arise since they are only targeting actual return statements. Protections are not being added to instructions that contain “bad” bytes. These bad bytes could lead to return statements if the instruction is started at a different location with languages with variable instruction sizes. This means that specific ROP gadgets would remain in the binary and, if newer methods are discovered to bypass the tools protections, the ROP gadgets they were protecting could still be used. Runtime-based protections are used during the runtime of the program. One method involved implementing a counter for how many returns were used in a specific number of instructions. If the counter number exceeds a certain amount, the execution of the program halts. However, this method is not extremely efficient because runtime-based protections add an overhead to the execution of the program due to the need to check for violations to the protections during execution. In contrast, the method chosen in this research implemented one static binary rewriting phase and thus has a lower overhead during execution because it does not need to perform the runtime checks.

Significance of the Study

The goal of this study was to create a static binary rewriting tool that removes ROP gadgets that contain the byte “c3”, the return instruction for Intel, in ELF x86-64 binaries. It

did not target all instances of that “bad” byte, but instead targeted a subset of instructions that contained the “c3” byte and replaced them with bytes that do not lead to return instructions. However, while it was not able to fix all ROP gadgets, any ELF x86-64 binaries should be able to support the list of instructions used for fixing the ROP gadgets. Also, while the binary rewriter itself is not able to target Windows systems, the created instruction list should also work on Windows binaries since they currently use the Intel architecture like Linux. In other words, the list of safe instructions for specific ROP gadgets should be applicable to any binary that uses the Intel architecture. Additionally, the tool does not require the source code of the binary for it to be patched. This, combined with the large target space of the tool, helps immensely with the security of software. It helps security in three ways. First, it allows users to patch systems themselves and not have to wait for vendor updates. Also, it allows for older software to be patched without needing it to be recompiled and allows for patching software even if the source code is lost.

For the rewriter side of things, the goal was not to create a “better” rewriter, but to simply offer another way of rewriting binaries to be investigated. The goal was to see how little extra information is needed to properly patch a large sample size of binaries. At the time of this research, the only extra information extracted is header information, section location and size, jump information, call information, relocation information and the full list of instructions for the binary. This excludes information such as function boundaries and a control flow graph of the binary. By gathering as little data as possible, larger binaries can be parsed due to needing to store less information about the binary.

Nature of the Study

The research method used for the study was DSR. The reason for this was that DSR attempts to solve a specific problem by creating an artifact that will solve the problem. The goal of the study was to produce two artifacts:

1. A static binary rewriter
2. A list of instruction replacements for ROP generating instructions

These artifacts combined would be used to reduce the amount of ROP gadgets in binaries. This also fits into a design problem classification described by Wieringa (Wieringa, 2014). A design problem attempts to invoke change by designing a solution to a problem (Wieringa, 2014). In this study, the change was a reduction of ROP gadgets in binaries to aid in the security of users. Additionally, the design for this research was the two artifacts created to solve the problem. These reasons show that taking a DSR approach was optimal for this study.

Quantitative (Cárdenas, 2019) and qualitative (Sandelowski, 1995) approaches were studied to ensure that a DSR approach was still the best method. Quantitative approaches focus mainly on the results of the research (Cárdenas, 2019). They have a heavy focus on the comparison of metrics to showcase the research. The goal of this study, however, was to highlight methods to remove ROP gadgets from binaries. While numerical metrics will be used to prove that the design is sound, they were not the focus of the research. Qualitative approaches use open-ended research questions to guide their research (Sandelowski, 1995). For this study, a potential question could be “Can static binary rewriters remove ROP gadgets from binaries?” However, to answer a question such as this, a tool would need to be created to

test this research question. Creating such a tool would be more suited to a DSR approach. So, it was decided not to use quantitative or qualitative approaches when conducting this study.

Objectives of the Project

The objective of the project was to produce a static reassemblable disassembler that could remove ROP gadgets without needing access to function or basic block information for a given binary. The list of questions that were examined are as follows:

1. Can a static reassemblable disassembler be created that does not rely on determining function or basic block information?

A tool was developed that attempted to parse a binary while only needing to determine the jump, calls, and relocations of the binary. Testing was done on the successful number of binaries patched and the integrity of the programs after patching.

2. Can a list of safe instructions be generated for specific ROP gadgets?

For a subset of ROP gadgets, a list of instructions for each ROP gadget was created. Every instruction list for the ROP gadget must:

    a. Maintain the original instructions functionality.
    b. Remove any bytes that would lead to a potential ROP gadget.

3. Can these lists of instructions be inserted into a binary using static binary rewriting?

This question had two problems that needed to be addressed. The first was identifying what instructions were generating the ROP gadgets. The other was how to handle the insertion of the given instructions. In other words, does the instruction get completely removed or simply modified with new values?

If the created tool solved the above questions, it would confirm several assumptions. First, that a tool can be created that does not rely on function or basic block information. Second, ROP gadgets can be directly removed from a binary using static binary patching. Lastly, the list of instructions used to eliminate the ROP gadgets could be integrated with other static binary rewriting techniques.

Definitions

Return-Oriented Programming (ROP): A attack that leverages return statements to gain control of a target program (Bhat, 2019).

ROP Gadget: A small list of instructions that occur before a return statement.

Static Binary Rewriting: Modifying a binary without needing to run or load the binary into memory.

Trampoline Rewriting: A static binary rewriting technique that changes the binary by diverting control flow to the patches being made and then returns control flow to the patch location (Schulte et al., 2022).

Reassemblable Disassembler: A static binary rewriting technique that inserts patches directly at the patch location (Schulte et al., 2022).

Assumptions

This study had to make a variety of assumptions during its course. Most of the assumptions fell upon the tools used during the research. These assumptions start with the belief that Capstone properly disassembles the Intel x86-64 architecture (GitHub, 2013/2023). The only time that Capstone sometimes failed was if data was hit while parsing executable sections in the binary. The assumption was that no data would be encountered in the Coreutils

binaries. In the same vein, it was assumed that, for instruction generation, Keystone would properly translate the new assembly instructions into valid byte code (Keystone, n.d.). The next area of assumptions fell under ROP and ROP gadgets. First, it was assumed that the Coreutils library would provide a wide range of ROP gadgets to perform adequate testing (*Coreutils.Git - GNU Coreutils*, n.d.). With this test bed, it was assumed that the Ropper tool would grab all or most of the ROP gadgets in the target binary (sash, 2014).

Scope and Limitations

The tool only targets ELF Intel x86-64 binaries. Further reducing this, it only targets the ELF binaries found in the Coreutils package. This was done because Coreutils is found in most Linux distributions and Uroboros, Ramblr, and Ddisasm all used Coreutils in their testing (Flores-Montoya & Schulte, 2020; Wang et al., 2017; Wang et al., 2016). Also, since there is a large variety of instructions that can cause ROP gadgets, only a subset of ROP gadget generating instructions was targeted. For limitations, complete control flow recovery is, in the general case, undecidable (Evans et al., 2015). Also, there was a limitation that should Capstone encounter data when parsing instructions, it will halt execution. Since the main goal was ROP removal, that issue will be overlooked and, should a binary cause this, it was simply marked as a failure.

All testing for this study was performed in a virtualized Linux environment using Windows Subsystem for Linux. The Linux environment was Ubuntu 20.04.6. The installed version of Coreutils was 8.30-3ubuntu2. There could be variances with the statistics gathered about the Coreutils binaries if done in a different Linux environment and/or with a different version of Coreutils. Additionally with artifact #2, the order in which ROP gadgets are

removed will vary if not done in the same way as in this study. However, the same result should be the same if the same path was taken.

Chapter Summary

This chapter explained the significance of adding protections to ROP based vulnerabilities. It also highlighted some of the current methods of static binary rewriting. Additionally, this chapter explored techniques in current use to attempt to alleviate ROP gadgets in binaries. An examination into whether it was feasible to use static binary rewriting to implement compiler-based approaches to remove ROP gadgets in their entirety from a binary was made.

A design science research approach was used to undertake the question of whether a static binary rewriter can be used to remove ROP gadgets from a binary. An artifact that can perform static binary rewriting and remove the ROP gadgets was generated. Using the DSR method also ensures that the generated artifact was sound and solves the research questions.

Chapter two will discuss literature related to this topic. These topics will include trampolines, trampoline rewriters, reassemblable disassemblers, ROP, and ROP mitigation techniques. It will highlight existing tools in the current space and some areas that they could be improved on.

CHAPTER 2

RELATED WORK

Chapter 2 will explore the topics needed to gain an understanding of the methods of binary patching. It will highlight existing tools and the different methods of binary rewriting. These methods will include dynamic and static binary rewriting and the multitude of ways to handle the rewriting of the instructions (Schulte et al., 2022; Wenzl et al., 2019).

After those techniques have been discussed, a deeper analysis will be taken on what ROP gadgets are and what mitigations have been put into place to hinder ROP gadgets. In addition, a look into how compilers and binary rewriting can be used as an aid to reduce the severity of ROP gadgets in binaries will be made.

Additionally, a comparison of the differences between Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) will be discussed. The main areas that differ that need to be considered are varying length instructions and the sharing of bytes in instructions. How these differences affect creating and hindering ROP gadgets will be shown in this chapter.

Reduced Instruction Set Computer Architecture

In regard to defeating ROP gadgets, one of the most important details about the RISC architecture is supporting a fixed instruction length (Fida El-Din & Krad, 2007). Having a fixed instruction length can limit the number of techniques for removing ROP gadgets. This can be seen in one of the approaches that E9patch (Duck et al., 2020) utilizes that involves using instruction punning. Instruction punning involves changing the leading bytes of an instruction so that the following bytes will become a new instruction of larger length. This

allows E9patch to have a wider range of location targets when creating new jump statements (Duck et al., 2020). This approach would not be as feasible in RISC architecture.

However, one benefit of having a fixed instruction length is that the number of potential ROP gadgets that can be generated is minimized. This is because for every potential ROP gadget location, the number of potential instructions for that location will be capped based on the defined instruction size. Also, when control flow is directed to a location, the location needs to be a valid instruction. With a fixed instruction length, the number of bytes that need to be correct is potentially less than in CISC.

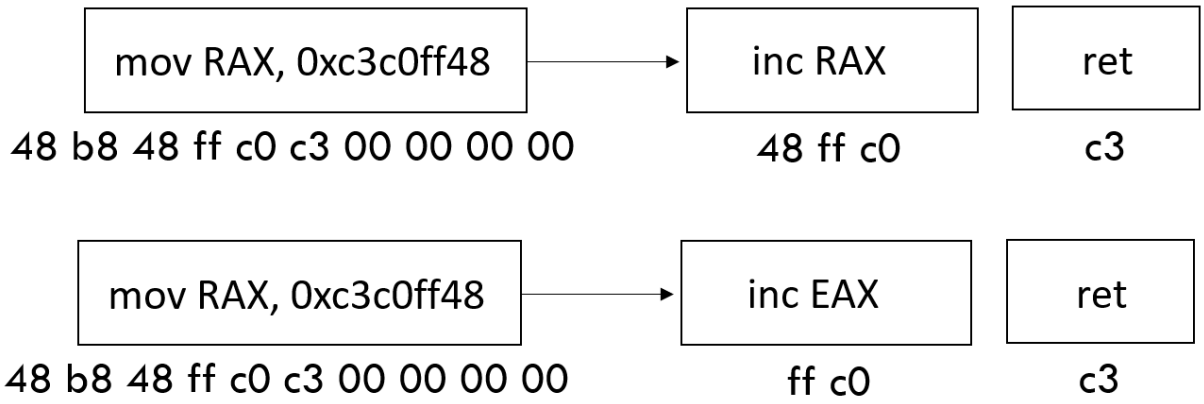
The overall focus of the tool created was to target ROP gadgets that were created by splitting instructions that contained a “c3” byte in them for Intel. In Arm, a comparable instruction would be a branch instruction. One example would be “bx r0” which is made up of the bytes “10 ff 2f e1” in 32-bit ARM. This would mean creating a mid-instruction ROP gadget would require four concurrent bytes instead of just one. Additionally, the total amount of potential gadgets would be lower because of the fixed instruction size.

Complex Instruction Set Computer Architecture

CISC architecture supports variable length instructions. This variable length of instructions can help create more ROP gadgets in binaries. The variable length allows for potentially more ROP gadgets at every return statement and an increase in valid return statements. Figure 3 showcases both problems. The first issue is that in the Intel architecture, a CISC architecture, the return statement is the byte “c3”. So, any instruction that contains the byte “c3” has the potential to create a return statement. However, a branch instruction in Arm 32-bit can be made up of the four bytes “10 ff 2f e1”. Only having to match one byte instead of four increases the number of potential return statements in a binary. The second issue is the

total amount of instructions that can be generated at a return location. The variable length of instructions allows for a larger variety of ROP gadgets created at a specific return. In Figure 3, just by modifying one instruction's bytes, two ROP gadgets can be created. If control flow of the binary can be started at the second "48" in the original instruction "mov RAX, 0xc3c0ff48", a "inc RAX; ret" gadget will be created. Additionally, if control flow can be started at the "ff" byte in the original instruction, a "inc EAX; ret" gadget will be created.

Figure 3 CISC ROP gadget creation



So, the CISC Intel architecture poses a larger risk than RISC architecture. This is because the return statement in Intel is only one byte. This leads to a larger number of return statements in binaries. Additionally, the number of ROP gadgets created at these locations in Intel has the chance to be higher than a RISC architecture implementation. For both of these reasons, the Intel architecture was chosen for this study.

Static Binary Rewriting

Static binary rewriting involves making all modifications to the binary without it having to be run or loaded into memory. A large number of differing tools have been

proposed for static binary rewriting (Schulte et al., 2022; Wenzl et al., 2019). The tools investigated can be divided into two categories:

1. Trampoline Rewriting
2. Reassemblable Disassembler

The trampoline rewriters involve adding additional instrumentation when trying to insert patch instructions. The reassemblable disassemblers directly replace instructions for their insertion method. Due to the nature of removing ROP gadgets directly, taking a reassemblable disassembler approach seemed to be the best option.

Trampoline Rewriting

Trampolines (Bernat & Miller, 2011; Kang, 2017), in the context of binary patching, are a three-step process. First, they require the ability to modify the control flow of the program. This usually entails the ability to insert a jump statement at specific locations, but any control flow modification technique will suffice. Second, they must be able to insert code/instructions at the location the control flow has been redirected to. Lastly, they must be able to return to the original location of where the control flow was redirected from.

The value behind being able to create these trampolines is maintaining the size of a section for a given binary. If the size of the section stays the same, section headers and locations will not need to be updated. So, if the programmer can create a trampoline in a "safe" location, it can avoid the risk of affecting existing sections. Once in this location, a large number of instructions can be inserted without the worry of having to modify the existing binaries headers and jump tables.

There are three main reasons why the created tool avoided using trampolines. The first had to do with memory fragmentation (Randell, 1969; Soma et al., 2014). Memory

fragmentation can be caused due to the nature of how trampolines allow the newly inserted instructions to be placed anywhere that is a safe location. This pseudo-randomly assigned insertion of instructions can lead to high memory fragmentation (Duck et al., 2020). Secondly, looking at trampolines from a ROP removal standpoint, the number of valid bytes when trying to construct a trampoline without introducing a new ROP gadget will be reduced. This is due to the fact that certain bytes that could be used for control flow redirecting, such as the "c3" byte, will need to be avoided. Lastly, trampolines need to add additional instructions for insertions for the jump structure. These additional instructions can start adding up if a lot of insertions are made. Also, for small insertions or even one-byte insertions the overhead is extremely large.

Safe Memory Location

One current technique, at the time of this study, for binary patching involves finding safe memory locations to insert new code instructions. For most cases, a safe location is one not occupied by a trampoline and is not within the .text or .data sections (Duck et al., 2020). The .text section contains the original binary code. If the new insertions are not made in the .text section, the need to worry about shifting the original control flow instructions will be eliminated. The insertions will need to be kept out of the .data section as well so that any reference to the data will not need to be altered due to the insertions. A safe location also needs to be a valid address and not NULL or a negative one (Duck et al., 2020). Attempting to insert at an address that is NULL or negative would not be contained in a binary's address space. If an insertion was made at either a NULL or negative address, the execution of the program would fail once the trampoline was reached. If none of these trampoline conditions are met, it should mean that instructions can be inserted at that specific address. This allows

for the target instruction being patched to be transformed into a jump to the newly found safe address. By doing this, the size of the .text section will not be increased, and the offsets should not have to be changed.

Trampoline Rewriters

Trampoline rewriting involves using some form of control flow diversion to "jump" to a location with newly added instructions and then "jump" back to the original patch location without needing to shift the original code base (Bernat & Miller, 2011). Four tools, BinPatch (Hu et al., 2019), Embroidery (Zhang et al., 2017), E9Patch (Duck et al., 2020), and Multiverse (Bauman et al., 2018) were studied to gain an understanding of differing methods of current trampoline patchers.

BinPatch is used to remove vulnerabilities from binaries based on existing patches for those vulnerabilities (Hu et al., 2019). BinPatch uses binary comparisons of existing functions to determine the locations of vulnerable functions within the binary (Hu et al., 2019). This is done to determine where the tool will need to insert trampolines to fix the vulnerabilities in the found functions (Hu et al., 2019). An issue with this approach is that all functions that may be used for comparisons will need to be stored somewhere or looked up to attempt to find a match when searching for vulnerable functions. When a vulnerable function is found, a trampoline will be created in the function and will contain the patched instructions based on a known fix (Hu et al., 2019). This means that to perform a fix for a vulnerability, there must be a known fix for that specific vulnerability.

Embroidery employs a similar patching method used by BinPatch (Zhang et al., 2017). They use the Android Security Bulletin to find vulnerabilities and use pattern matching to find vulnerable functions in select binaries that were found in the bulletin (Zhang et al., 2017).

This tool has one of the same limitations as BinPatch, which is that a fix for a vulnerable function must be known to apply a patch to a binary. Additionally, Embroidery is designed for the Android ecosystem (Zhang et al., 2017). This study was designed for Linux distributions. So, a large number of modifications would need to be made to even attempt to use this tool for ROP gadget removal for a Linux environment.

Multiverse uses a superset disassembler and creates a new section called .newtext to perform trampoline rewriting (Bauman et al., 2018). Multiverse attempts to brute force every executable byte offset to determine a superset of the code for patching (Bauman et al., 2018). To guarantee safe locations in the binary, the .newtext section is used (Bauman et al., 2018). A mapper links the superset to the .newtext section to determine addresses for the trampolines (Bauman et al., 2018). It also creates new mappings for user defined functions to attempt to further enforce the integrity of the program (Bauman et al., 2018). There were two main reasons this approach was not taken for this research. The first was the tool was designed for the x86 space and not the x64 space of instructions. The second reason was because of the insertion of the new .text section into the binary. Since Multiverse adds the new .text section and keeps the old .text section just with added trampolines, the size of the patched binaries is greatly increased (Schulte et al., 2022).

E9Patch allows for control of the instructions being inserted rather than basing them off an existing patch. E9Patch leverages the ability of Intel (Turley, 2014) instruction punning (Chamith et al., 2017) to create trampolines (Duck et al., 2020). Unlike RISC architecture (Patterson & Sequin, 1981), Intel supports variable size instructions. This means that each instruction is not the same length. This will be a key factor when attempting to insert instructions into a binary or modify it because a modified instruction could change its length

after a patching attempt. Instruction punning allows for the same bytes to be used for multiple instructions. This can be used to save memory in the program if instructions are allowed to share those same bytes. The ability to use instruction punning allows for a greater set of possible valid trampoline targets. In addition, E9Patch also uses the variable instruction length to create the first of their three patching tactics: Padded Jumps (Duck et al., 2020). This method considers if the instruction being modified is not five bytes in length, which is the size of larger jumps. If it is not five bytes, it will need to be padded out to reach those 5 bytes.

The other two techniques they offer, Successor and Neighbor Eviction, involve adding near-by instructions into the patch location and using them to create mini trampolines as needed (Duck et al., 2020). The main issue with reusing bytes of the original instructions for creating trampolines for ROP gadget removal is it further reduces the valid number of bytes that can be used. In this study, only the “c3” byte was targeted for ROP gadget removal. This would mean that the “c3” byte would not be able to be used to create new trampolines. However, there are other bytes that can generate ROP gadgets and each one that is added would also constrict the number of bytes that could be used.

While trampoline rewriting is generally more successful for implementing patches, they tend to increase the runtime of the binary more than reassemblable disassemblers (Schulte et al., 2022). Another issue caused by trampoline rewriting is high physical and virtual memory fragmentation (Duck et al., 2020). Also, should a ROP mitigation technique be implemented, it would reduce the number of bytes that are able to be used when trying to find a safe location when creating the trampoline. Because of all of these considerations, it seemed that taking a reassemblable disassembler approach was better for removing ROP gadgets from existing binaries.

Reassemblable Disassemblers

Unlike the trampoline rewriters, reassemblable disassemblers insert the patches directly into the instruction set rather than using trampolines (Wang et al., 2016). This, in theory, should mean that the binary size increase from insertions should be less because the trampoline instructions do not have to be added. It should also not run the risk of needing a "safe" location for the trampoline locations. Four tools were analyzed for this: Uroboros (Wang et al., 2016), Ramblr (Wang et al., 2017), Ddisasm (Flores-Montoya & Schulte, 2020), and RetroWrite (Dinesh et al., 2020).

Uroboros was one of the original reassemblable disassemblers created (Wang et al., 2016). Uroboros implemented an advanced linear sweep method called BinCFI (Wang et al., 2016). BinCFI uses a standard linear sweep approach for the first pass of a binary, but checks for errors or gaps in the produced output (Zhang & Sekar, 2013). Errors are defined as instructions that contain invalid opcodes, direct control transfers outside the current module, and direct control transfers to the middle of an instruction (Zhang & Sekar, 2013). The errors are caused by gaps in the program which are found by locating improperly disassembled instructions (Zhang & Sekar, 2013). To determine the size of the gap, a backwards search for the nearest unconditional control-flow transfer is made (Zhang & Sekar, 2013). Once every gap has been marked, another disassembly pass will be made over the entire binary ignoring any gap locations (Zhang & Sekar, 2013). The entire process of marking gaps and fully disassembling the binary will be repeated until no errors and gaps occur (Zhang & Sekar, 2013).

This study implemented a linear sweep algorithm but used less error checking than BinCFI when pulling out instructions. Additionally, the binary was only disassembled one

time rather than multiple times. Also, the only time in this study a backwards propagation method was used was when attempting to fix jump table instructions.

Uroboros leverages BinCFI to attempt to build out basic block information and function information during its disassembly phase and stores the results for rewriting later (Wang et al., 2016). The results are also used to build out the control flow graph for the target binary (Wang et al., 2016). However, the tool does not perform well compared to newer disassembler rewriters so other tools were analyzed (Schulte et al., 2022).

Ramblr improved upon Uroboros' basic block detection. It uses Angr (GitHub, 2015/2023) and recursive traversal (Kinder, 2010) to build out the entire control flow graph of the target binary. Angr is a tool that uses both static and dynamic symbolic execution (King, 1976) for analysis of programs (GitHub, 2015/2023). Ramblr builds upon the symbolic execution provided by Angr and uses recursive traversal methods to ensure that the control flow graph is complete (Wang et al., 2017). Ramblr also uses multiple techniques to attempt to classify what type the value actually represents (Wang et al., 2017). Some of those types include primitives, strings, jump tables, and arrays of primitives (Wang et al., 2017). Since Ramblr uses Angr, the user is also required to have Angr as a dependency. It also means that if Angr cannot parse the control graph of the target binary, Ramblr will fail to function.

One of the goals of this study was to create a static binary rewriter that did not need to symbolize the binary or build out basic block information. This was done to try to decrease the amount of overhead the tool needed to function. Since Ramblr is built upon Angr, that is already additional overhead. Furthermore, it builds out the entire control flow graph, which is what this research was trying to avoid.

Unlike Ramblr, RetroWrite uses a linear sweep method to determine the control flow graph of the target binary (Dinesh et al., 2020). The most notable part of RetroWrite is the symbolization that it performs against the target binary. The symbolization process is broken into three different steps. The first step of the symbolization process is the control flow symbolization (Dinesh et al., 2020). In this step, control flow instructions are converted to assembler labels (Dinesh et al., 2020). The tool created in this study stores additional data related to control flow instructions but does not build out a label for the reference. The second step is converting PC-relative addresses (Dinesh et al., 2020). During this step, instructions using offsets with the RIP register will have their location converted to a assembler label (Dinesh et al., 2020). Additionally, the instruction will use this label as its new reference location instead of using just RIP and an offset amount (Dinesh et al., 2020). These types of instructions were handled in a similar fashion in this study. The difference was only the offset amount was saved and used for calculations during the rebuilding process. No labels were made for the location of the target address. The third step was data relocations (Dinesh et al., 2020). In this step, any data reference had the bytes that were being targeted converted to an assembler label that would be referenced by the original data location (Dinesh et al., 2020). After this final step, all the symbolizations for the binary should be completed (Dinesh et al., 2020).

Once all the symbols have been gathered, RetroWrite will write back to an assembly file adding all of the symbols to this temporary assembly file (Dinesh et al., 2020). This allows other tools to make modifications to the temporary assembly file should they require the labels. To determine the disassembly to make this temporary assembly file, RetroWrite uses Capstone to parse the assembly instructions from the byte code (Dinesh et al., 2020).

This approach was similar to the approach used for this study, except the goal was to do this without needing to make new labels and simply keeping a list of all relocations. Additionally, the tool created for this study does not lift the binary into a temporary assembly file like RetroWrite.

Ddisasm is one of the best performing binary rewriters (Schulte et al., 2022). It leverages Capstone (GitHub, 2013/2023) to generate the instruction set. One unique feature is that it uses the Datalog language (Margaret, 2015). Ddisasm uses a linear sweep method with Capstone until it reaches an invalid instruction that Capstone cannot handle (Flores-Montoya & Schulte, 2020). Should that situation happen, Ddisasm will employ a backward propagation technique and a forward traversal technique to attempt to resolve the instruction (Flores-Montoya & Schulte, 2020). Similar to Ramblr and Uroboros, Ddisasm attempts to build code blocks after parsing the instruction set (Flores-Montoya & Schulte, 2020). However, Ddisasm employs a more advanced code block detection strategy that also attempts to solve block conflicts if they are encountered when being built (Flores-Montoya & Schulte, 2020). Since Ddisasm had one of the best success rates of the static binary rewriters, the goal of the research was to attempt to get as close as possible to the same success rate. However, this would be done without building out the basic block information that Ddisasm uses to perform its static binary rewriting.

Generating ROP Gadgets

The Intel architecture allows for a variety of methods to determine how the bytes in the binary are used. They can be shared with instruction punning and the starting point of an instruction can be changed with a control flow operation. This allows an attacker to create ROP gadgets even if there was originally no return in the instruction set of the binary

(Shacham, 2007). This method involves searching for “bad” bytes such as a “c3” byte, which is the op code for a return instruction in Intel. Since instructions can share bytes in the Intel architecture, this reuse of bytes can be used to create ROP gadgets that were not created by a standard return. This is done by taking an instruction that contains a bad byte in it and then converting control flow to start at an address that will treat that bad byte as a type of return instruction. So, if the malicious user has control of the execution of the program, new ROP gadgets can be generated from instructions that originally contained no ROP gadgets.

ROP Mitigation Methods

There have been previous studies on how to mitigate the impact of ROP gadgets in a given binary. The methods can be classified into the following approaches: compiler-based mitigations, return-based mitigations, and dynamic-based mitigations (Ruan et al., 2016). These methods and tools for them will be described in the following paragraphs.

One technique used to try to reduce the impact of ROP gadgets for a program is to try to remove instructions that result in potential ROP gadgets during compilation time. One such tool that does this is G-Free (Onarlioglu et al., 2010). G-Free attempts to identify bad instructions when a program is being compiled. These mitigations involve shifting jumps, changing what registers are used, and adding cookies to new return instructions (Onarlioglu et al., 2010).

Some problems with compiler-based approaches are:

1. Some compilers will not analyze assembling instructions and simply link them (Newline, 2021).
2. Requirement of the creator of the executable binary to have used the compiler mitigation.

3. If the creator did not use the mitigation, the creator must provide the user source code with the binary to allow the user to perform the mitigations themselves.

The first issue means that if any in-line assembly is used, G-Free will not be able to detect it because it will never be seen. This issue can be exasperated if a library function that programs use contain in-line assembly. The second issue forces the requirement of using G-Free onto the developers of programs. This is burdensome to the developer and puts all the responsibilities onto the developer. Additionally, end users would not be able to patch compiled programs and must rely fully on the developer to secure the program. For the last issue, most publishers of applications will not supply source code along with their products. This means that users would not be able to use the complier-based approach at all. Because of these concerns, taking a compiler approach is not optimal for end-users. However, this study did attempt to integrate some of the techniques showcased in G-Free in a binary rewriting approach. The techniques implemented involved the removal of instructions causing gadgets and replacing them with instructions that were equivalent but did not contain the “c3” byte.

Another technique to hinder ROP gadgets is to use binary rewriting to insert additional instructions or canaries around functions and their returns. Three tools that do this to some degree are RAD rewriting (Prasad & Chiueh, 2003), AT-ROP (Xu et al., 2020), and ret_ROP (Xu & Wang, 2022). All these tools involve some method of adding instructions around a potential ROP gadget. One issue with these tools is that they look for standard return statements and not the ones generated from created ROP gadgets.

RAD rewriting adds RAD (Chiueh & Hsu, 2001) code at the beginning and end of specific functions to hinder ROP (Prasad & Chiueh, 2003). To accomplish this, RAD rewriting attempts to disassemble a binary and detect function boundaries (Prasad & Chiueh,

2003). Only functions deemed as “interesting”, which is described as functions that use stack frame allocation and deallocation of local variables, are targeted by RAD rewriting (Prasad & Chiueh, 2003). Each interesting function will need to have additional code assigned to their prologues and epilogues (Prasad & Chiueh, 2003). The prologue will gain additional code that saves a copy of the return address in the return address repository and the epilogue will gain additional code that will check the value in the return address repository when the function attempts to return (Prasad & Chiueh, 2003). If the two return values do not match, the program will terminate execution (Prasad & Chiueh, 2003). To allow space for the additional code, a new section will be appended to the end of the original binary (Prasad & Chiueh, 2003). Each function will have jump instructions to the correlating RAD instructions in the new section (Prasad & Chiueh, 2003).

One issue with this approach is the need to define each function when attempting to disassemble the binary. If a function is not found, a potential vulnerability could remain in the program. Another issue is that adding an entirely new section increases the size of the binary. Regarding this study, the tool created does not define function boundaries and could not implement this technique. Also, RAD rewriting would not fix ROP gadgets caused from instruction splitting. The tool created in this study specifically is designed to fix the mid-instruction ROP gadgets.

AT-ROP attempts to hinder ROP gadgets by clearing function parameters before function returns (Xu et al., 2020). To accomplish this, AT-ROP builds a control flow graph of the target binary during a disassembly phase and defines function boundaries (Xu et al., 2020). For each function, AT-ROP marks ret_blocks that contain the instructions leading up to the return and the return statement (Xu et al., 2020). The ret_block will be converted to a

trampoline that changes control to a new section that AT-ROP creates that contains ori_data and clear_data (Xu et al., 2020). ori_data contains the original instructions in the ret_block and clear_data contains the instructions: “xor rdi, rdi; xor rsi, rsi; xor rcx, rcx; retn” (Xu et al., 2020). These commands will clear out the first, second and fourth parameters of the function (Xu et al., 2020). Clearing the registers can help stop ROP gadgets that perform operations such as “pop rdi; ret” (Xu et al., 2020). This method requires the determination of function boundaries, which is something that the tool created in this study is not able to do because of the constraint of no symbolization. Also, having to create a new section as valid trampoline targets introduces additional overhead that a reassemblable disassembler does not have. In addition, AT-ROP’s mitigations do not target instructions that generate ROP gadgets by starting the instruction at a different offset. The tool created in this study was designed exclusively to target those types of instructions.

ret_ROP adds instructions that will clear function parameters before function returns (Xu & Wang, 2022). The instructions are added using static binary rewriting (Xu & Wang, 2022). To determine function boundaries, a control flow graph is built with the aid of Angr (Xu & Wang, 2022). Similar to AT-ROP, ret_ROP defines a ret_node as a sequence of instructions at the end of a function that end with a ret instruction (Xu & Wang, 2022). The size of ret_node will be checked to ensure that the total length is at least a size of five bytes (Xu & Wang, 2022). The length is checked because a trampoline will be inserted at the ret_node and the size needed to construct a far jump is five bytes (Xu & Wang, 2022). The trampoline target will be in a new program segment that ret_ROP adds (Xu & Wang, 2022). The instructions inserted at the trampoline target will be the original ret_node instructions plus the addition of three instructions that will clear the first, second, and fourth parameters of

the function (Xu & Wang, 2022). Like AT-ROP, the addition of new segments increases the size of the binary, but the tool created in this study does not. Also, ret_ROP's method of patching involves building out a control flow graph of the target binary. The tool created for this research does not build out a control flow graph and thus could not implement this type of ROP defense. Additionally, the approach taken in ret_ROP would also not hinder ROP gadgets that are created from mid-instruction bad bytes.

The final method is trying to detect ROP gadgets being chained during runtime. One such tool that does this is ROPecker (CHENG et al., 2014). ROPecker implements a sliding window that keeps a tally of how many potential ROP gadgets have been executed recently and, should too many be detected, the program will stop execution (CHENG et al., 2014). This effectively will disallow an attacker to build successful chains with this implementation. The main issue with taking a runtime-based approach is the additional overhead of having to check instructions during the runtime of the program. The goal of the tool was to eliminate ROP gadgets so that runtime checks would not be needed.

Most of the current methods introduced some form of additional overhead, whether that is a size increase from needing to add additional sections or segments like AT-ROP and ret_ROP or from performance overhead from runtime checks from ROPecker. Also, compiler-based methods require source code which most end users will not have available to them. The tool created for this research attempted to use a reassemblable disassembler approach to not have to add additional sections unless necessary. Additionally, if enough ROP gadgets were removed, the amount of feasible ROP chains should be reduced, and no runtime checks should need to be made. Lastly, the need for source code is not needed for the approach taken in this study.

CHAPTER 3

RESEARCH METHODS

Chapter 2 discussed current literature related to the topic area of this research. It highlighted current methods of performing static binary rewriting and different tools that were designed to perform those different methods. Additionally, it explained what ROP was and the impact ROP gadgets have. Lastly, it showcased ROP mitigation techniques and how they could be linked to static binary rewriting. Chapter 3 will detail the research methods that were used for this research. This chapter will explain why DSR was used for this study and how it abides by the research guidelines of both Hevner and Wieringa (Hevner et al., 2004) (Wieringa, 2014).

As stated in Chapter 1, this research attempted to design a tool that could integrate ROP gadget removal with a static binary rewriting method. The ROP gadgets that were targeted were ones created from address splitting and not standard return instructions. The ROP gadgets were chosen to fill the gap of research of only targeting ROP gadgets in standard functions with static binary rewriting.

Research Methods

Design science methodology was employed for this research and the artifact created, the static binary rewriter, answers the research question. Quantitative analysis (Cárdenas, 2019) places a large emphasis on the comparison of values and metrics defined for the research. While there are metric comparisons when testing the results of the tool, the process and design of the tool created was the emphasis of the research, not the metrics. Qualitative analysis (Sandelowski, 1995) uses open-ended questions for performing research. The

designed tool has defined metrics and a design process for creating a tool. There are no open-ended questions for this study. All these reasons show that both quantitative and qualitative approaches were not well suited for this study.

For the DSR approach, both Hevner's guidelines (Hevner et al., 2004) and Wieringa's methods (Wieringa, 2014) were followed. Hevner's work was used because of the range of impact his work has had on the design science space. However, his work falls under Information Science rather than under software development. Since the created tool lives under the software development and software security fields, Wieringa's approach was also followed. This was because Wieringa's approach was created with software engineering in mind (Wieringa, 2014). In addition to belonging to the software development space, Wieringa's approach is a highly technical way of performing DSR (Wieringa, 2014). Due to this technical nature, it is more applicable when dealing with specialized problems, such as removing ROP gadgets at the assembly level, than Hevner's work. Because of these reasons, Wieringa's methods were more suited to this research.

To abide by Wieringa's approach, the artifact created must be designed to improve the space it is being designed for (Wieringa, 2014). Two important distinctions are derived from this requirement. First, it allows the researcher freedom to develop a tool in a flexible manner to meet those improvements. Secondly, however, it also means that the improvements that were made must be clearly defined and can be properly validated. Wieringa states that design and validation are part of an iterative cycle (Wieringa, 2014). The design phase entails specifying the requirements of the proposed treatment and determining if those requirements will solve the goals of the research (Wieringa, 2014). The validation phase ensures that the

requirements were met and had the desired effects (Wieringa, 2014). These two phases will be included to ensure clearly defined improvements and proper validation.

Since the end goal was to develop a tool that could answer the research questions, Hevner's guidelines served as strong guides. This was because Hevner states that DSR is, at its core, a problem-solving method (vom Brocke et al., 2020). Following his guidance allowed for a strong artifact to be created that was developed specifically for the problem that was stated. So, by following both Wieringa's and Hevner's work, the strongest possible artifact was created.

Hevner's Guidelines

Hevner defined seven guidelines to be followed when performing design science research (Hevner et al., 2004). The seven guidelines are as follows:

I. *Design as an Artifact* – Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

II. *Problem Relevance* – The objective of design science research is to develop technology-based solutions to important and relevant business problems.

III. *Design Evaluation* – The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.

IV. *Research Contributions* – Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.

V. *Research Rigor* – Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.

VI. *Design as a Search Process* – The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.

VII. *Communication of Research* – Design science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

The following paragraphs will detail how the static binary rewriter abided by these seven guidelines. Also, as mentioned previously, some slight liberties were taken when examining them from a software design and software security viewpoint rather than an information science viewpoint.

To meet the first guideline, some implementation of a solution must be created. To meet this guideline, this study created a tool that implements the solutions outlined previously. The tool can be found at GitHub under the *Neptunia* repository created by *DSUHansVerhoeven*. This tool satisfies the requirement of creating a deliverable artifact and implementing an instantiation of the artifact.

When addressing the second guideline, the first liberty was taken. The guideline states that the technological solution must be designed for a relevant business problem. In the context of this study, this was shifted from a relevant business problem to a relevant cyber security problem. One of the most commonly exploited vulnerabilities in 2021 involved the use of ROP (Cybersecurity & Infrastructure Security Agency, 2021). Since it is still used in common vulnerabilities, the goal of trying to reduce the ability for ROP exploits to function is still an important problem. This can be seen as a fair conversion because the goal of design science is to address critical problems and should not solely be limited to the business scope.

Additionally, the first part of the guideline states that the objective is to create a technology-based solution. This is quite easy to abide by because the artifact is a Python based tool that integrates with current software tools.

The third guideline addresses the soundness of the evaluation of the model. As stated, it must be demonstrated using well-executed evaluation methods. The methods used for testing in this study were chosen based on what previous tools in this area of research used. For determining the success of the static binary rewriting functionality, the Coreutils test suite was used. Three current reassemblable disassemblers, Uroboros, Ramblr, and Ddisasm, all used this suite in their testing metrics (Flores-Montoya & Schulte, 2020; Wang et al., 2017). These tools were created and published in peer-reviewed works. Due to that, Coreutils can be deemed as a proper execution metric. Ddisasm also implemented a method of testing that involved inserting NOPs every certain number of instructions found in the target binary. This was done to highlight the ability to apply patches in any spot of the binary. This study also implemented this approach to demonstrate the capabilities of the created static binary rewriter. One of the survey papers also used increase of the binary size as a defining metric in the capabilities of the static binary rewriters (Schulte et al., 2022). The size of the binaries was checked before and after running the static binary rewriter. This allowed for only the static binary rewriter to have affected the size of the binary during testing. Additionally, the size of the binary was measured using built-in Linux tools. These native tools have provided core functionality to the Linux system and are open source for peer review (*Coreutils.Git - GNU Coreutils*, n.d.). Because of those reasons, they can be assumed to be valid.

For demonstrating effectiveness of the ROP removal portion of the artifact, a tool called Ropper (sash, 2014) was used. Ropper was chosen because it comes native on Kali

Linux and it is one of the more highly regarded ROP tools (Kali Linux Tools, n.d.; *ROP Emporium*, n.d.) This tool grabs the bytes of the binary and returns potential ROP gadgets. In this study, a measurement of the amount of ROP gadgets in the binary was taken before and after the tool was run against the binary. This returns the amount of ROP gadget reduction in the binary. By using all these methods, the artifact created properly followed the guidelines for design evaluation.

Hevner's fourth guideline states that the artifact created must have relevant contributions toward the targeted research space. The contributions of this research were threefold. First, it provided another look into the amount of information needed for applying static binary rewriting. Secondly, it provided methods for end users to secure their software without needing the developers to secure it beforehand. Lastly, it supplied a list of instructions that can be used to replace ROP gadgets to remove them from a binary. So, if better methods of patching are discovered or if a user wants to use a different binary rewriter, the list generated can still be used in those tools for replacements. This allows the artifact to affect all other binary rewriting techniques regarding removal of ROP gadgets. These three contributions mean that the created tool has made relevant contributions to the static binary rewriting and ROP defense spaces.

The fifth guideline defines the research rigor and its need. To achieve rigor, one must properly employ existing methodologies and basis (Hevner et al., 2004). This study followed a rigorous approach by abiding by Hevner's guidelines (Hevner et al., 2004) and Wieringa's approaches (Wieringa, 2014) and by using existing tools throughout the research. This rigor is needed not only in the construction of the artifact, but also in its testing (Hevner et al., 2004). For testing rigor, previously used tools and notable tools were used for testing. The rigor for

construction comes from the highly module design of the tool's construction and design research. This modular design allows for the tool to adapt to any needs to create a working artifact. A working artifact for the test space implies that proper research rigor was used in the creation and testing of the artifact.

The sixth guideline discusses the need to use existing methods and to satisfy any legal requirements in the problem space. Addressing the existing methods, the artifact used test cases and tools that current tools were using. The methods these tools use was also examined. Both of these factors address the utilizing available means section of this guideline. The legality of the problem space falls under the modification of binaries that have been supplied to end users. For the test corpus of Coreutils, these binaries are able to be modified without worry due to their open-source nature. Problems could arise when using the artifact on binaries that are not a part of this binary suite. Some vendors have licensing agreements on what users are allowed to modify about the program. The created tool makes assumptions about instructions and indiscriminately modifies them. This has the potential to violate the previously mentioned licensing agreements. However, this is outside the scope of the artifact and thus the artifact should satisfy the legality section of the guideline. With both sections satisfied, the artifact should meet Hevner's sixth guideline.

The last guideline involves the ability to communicate the findings to technologically and non-technologically oriented people. The artifact should satisfy both due to the nature of the results. The evaluation metrics are before and after results, which means that they can be produced in metric charts, which all people should be able to understand regardless of their technological background. The release of the list of instructions, while the instructions

themselves might not be understood by all, still showcases what was changed in the binary to an understanding of all.

This means that the artifact should meet all Hevner's guidelines for performing design science research. It also shows that even without being in an information science space, this research still abides by all the guidelines. In the following section, an introduction to the design science methods, introduced by Wieringa, employed to develop the artifact will be given.

Wieringa's Methods

Wieringa splits design science problems into two different categories: design problems and knowledge questions (Wieringa, 2014). Design problems attempt to solve problems by creating a design that has direct impacts on the space (Wieringa, 2014). Knowledge questions involve trying to determine an answer about a question for the chosen space (Wieringa, 2014). Since the nature of the artifact created was to design a tool to perform static binary analysis, it will fall under the category of a design problem.

Now, with the problem type decided, we can move to what Wierenga considers the design cycle or part II of design research problems. For design problems, he splits part II into three parts: problem investigation, treatment design, and treatment validation (Wieringa, 2014). The following sections will explain how each of these parts was used in this study.

Since the problem was classified as a design problem, the tool needed to complete all three sections in the design cycle. The first section to be addressed was the problem investigation step. During this step, the determination of whether the task can be completed, can an artifact be designed for the space, and will there be an impact on the space should the artifact be created needed to be decided. (Wieringa, 2014). Due to tools being created for

similar tasks in this space, it was reasonable to assume that the task should be able to be completed. In addition, the ways that the tasks have already been solved through the use of created tools means that the designed artifact is able to be generated for this space. Also, since ROP removal tools exist and more securities are being developed for ROP, in general, the artifact made has an impact on the space. This means that the design should completely abide by the problem investigation step.

The second step is treatment design. This step involves the methods used to solve the problems identified (Wieringa, 2014). As mentioned in earlier chapters, the treatments used are similar methods that existing tools use, but specific aspects of them were changed. This means that the artifact should still be able to solve the problem in the chosen space but do so in a different manner than existing solutions.

The last step is treatment validation. Validation is ensuring that the design will solve the problem it was created for (Wieringa, 2014). For this step, the artifact generated must be validated and it must be ensured to have solved the problem (Wieringa, 2014). To perform this task, the before and after approach mentioned earlier in chapter 3 was used. This involved a before and after testing phase of target binaries with the tool run against them. The tools used for analysis and the methods have been used by existing tools previously, so they are technically sound metrics. Once the metrics were gathered, the results were analyzed to see if the treatment of the problem space was met. These results will be discussed in a later chapter.

These sections described what Wieringa's methods were and how this study followed those methods. This means that this research followed not only Wieringa's methods, but also Hevner's guidelines, indicating that this study followed a proper design science research approach.

Base Design of the Tool

The core design of the tool did not start from scratch. The base tool had previously been designed in conjunction with Logan Stratton, a staff member at Dakota State University. The tool originally investigated was designed to be a GCC plugin for removing ROP gadgets. However, there were some issues that were found. The issues were mainly from preassembled instructions not being targeted by the plugin. This changed the approach to a beta version of this tool. He, however, stepped away from the tool during the creation, but this study still used material he provided. The materials he provided will be detailed below.

The first area he provided code for was the definition of classes for specific sections and the call and relocation structures. These classes are used throughout the tool when rewriting the calls and relocations. He also provided functionality for determining the section an instruction belonged to and if the section was executable when parsing. The largest contribution he made was handling the reassembly of the target sections based on the modified instruction set that was generated during the runtime of the tool.

However, some modifications needed to be made to his code base. First, his logic for shifting the calls and relocations was incorrect in some cases. These cases involved the improper use of negative relocations. His methods were improperly grabbing negative values for relocations. So, his logic for shifting relocations was only designed properly for positive values. This issue was solved by rectifying the issue of not grabbing negative values properly and then adding proper logic for negative values. In addition to this, when a two-byte jump is shifted to a five-byte jump due to insertions, he was not shifting the calls and relocations by this increase, which caused the created binary to fail. No issues have been encountered with his reassembly of the binary thus far and his work is extremely appreciated.

Chapter Summary

This chapter has discussed how a design science research methodology was proper for this study. It described what Hevner's guidelines and Wieringa's methods were. Additionally, it detailed how this study abided by Hevner's guidelines and Wieringa's methods on design science. Further information about the details of the population and the sample of the study was also provided. Lastly, information about the how the tools being used in this study also abide by the guidelines and methods for design science was given.

The next chapter will explain the design plan used to create both artifacts for this study. Additional information about how data was collected and analyzed will also be mentioned. Finally, the next chapter will also define the scope and metrics that were used for this research.

CHAPTER 4

DESIGN AND IMPLEMENTATION

Chapter 3 listed how the artifacts being designed would abide by Hevner's guidelines and Wieringa's methods. Chapter 4 will give a description of what the overall objectives for the artifacts are and how the artifacts were designed. Information about the population and sample chosen for this study will be provided in this chapter. Also, the reliability and validity of both the data collected and the tools used will be discussed. Lastly, how the data was analyzed to provide the metrics for the results is provided in this chapter.

Artifact Objectives

The goals of the artifact were split into two areas. The first area involves the goals for the static binary rewriting techniques and the second is the goals for the ROP gadget removal methods. The goals for the static binary rewriting techniques are as follows:

I. The static binary rewriter will only parse the following information:
    a. The sections of the binary.
    b. The status of each section.
    c. The jumps located in the executable sections.
    d. The calls located in the executable sections.
    e. The relocations located in the executable sections.
    f. Each instruction located in the executable sections.

II. Information involving the above requirements will be stored in separate classes.

The goals for the ROP removal portion of the static binary rewriter are as follows:

III. All targeted ROP gadgets will be linked to the instruction that caused them.

IV. If the instruction causing the ROP gadget belongs to the subset of instructions being targeted, it will be replaced with the instruction list to destroy the ROP gadget.

V. A list of instructions generated for the ROP gadget removal will be released.

All five of these requirements must be met for the artifacts to be deemed successful.

Implementation

Since there are two artifacts being created, two design cycles were needed to create both artifacts. The first design cycle, shown in Figure 4, involved the creation of the static binary rewriter. The second design cycle, shown in Figure 5, involved modifying the artifact created from the first design cycle to remove ROP gadgets from the target binary.

For the first step in the first design cycle, the binary must be loaded into the artifact so parsing may occur. Once the artifact has been loaded, the sections of the binary will be located using pyelftools (Bendersky, 2013/2023). After the sections have been located, the flags each section contains will be used to determine if the section is executable or not. Each of these executable sections will have their instructions parsed using Capstone. In addition, the instructions containing jumps, calls, or relocations will also be stored in subsequent data structures. Once all this information about the binary has been gathered and stored, the insertion process can begin. The insertion process is a multistep process that is shown in Figure 6.

Figure 4. First design cycle

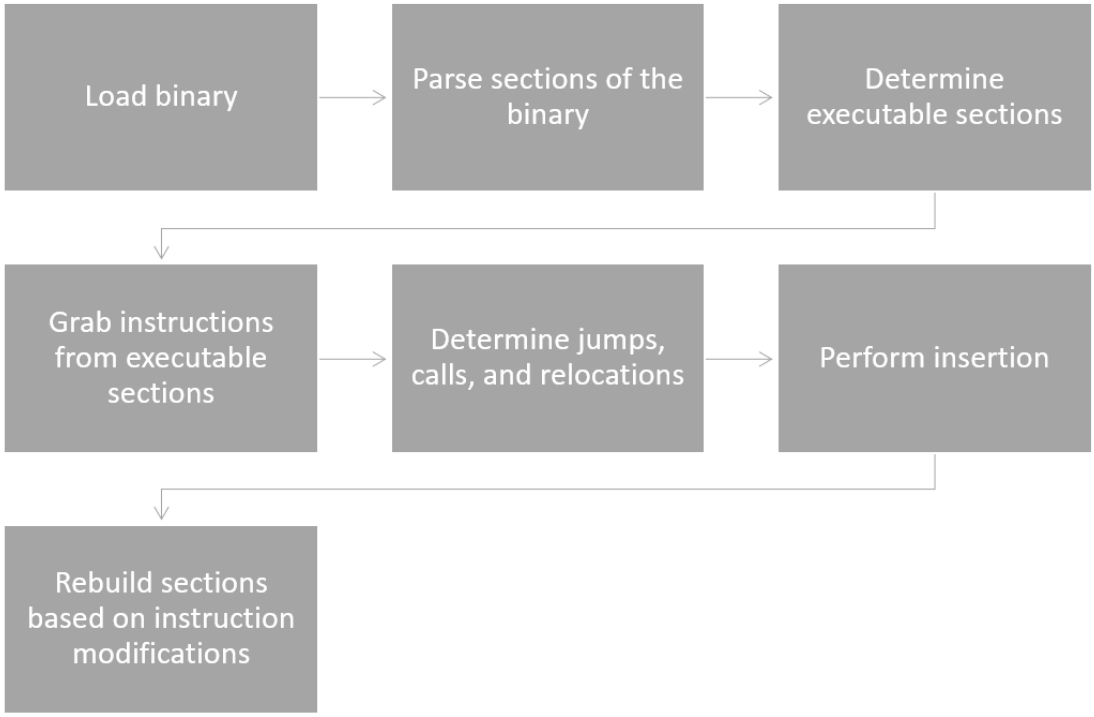


Figure 5. Second design cycle

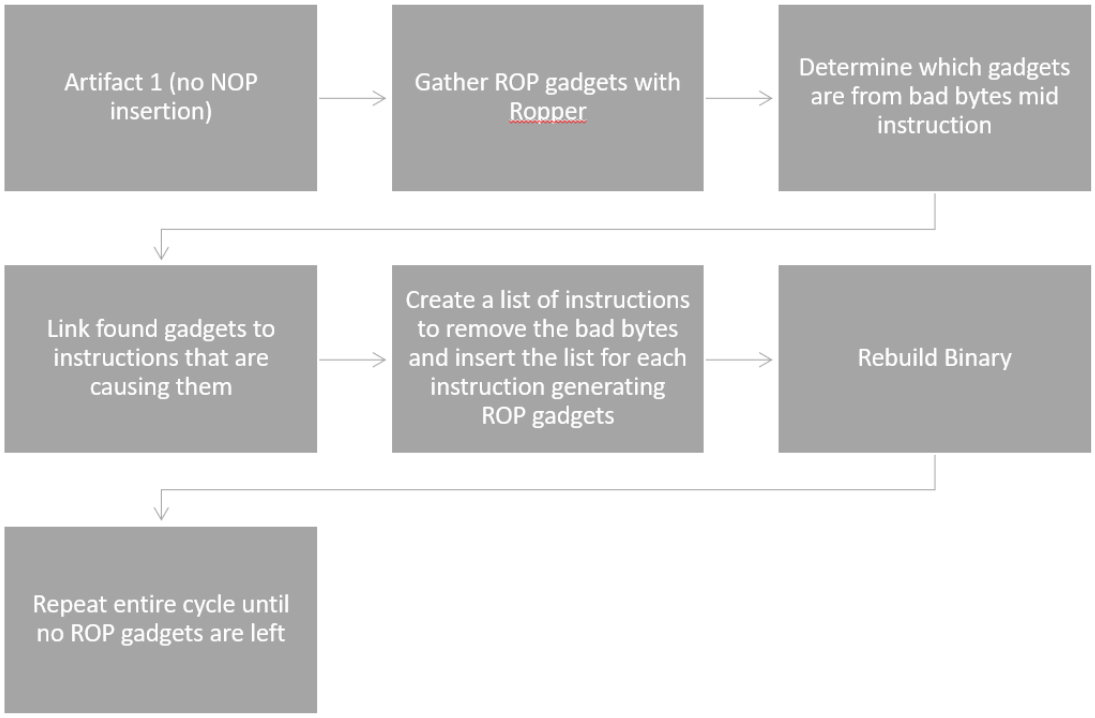
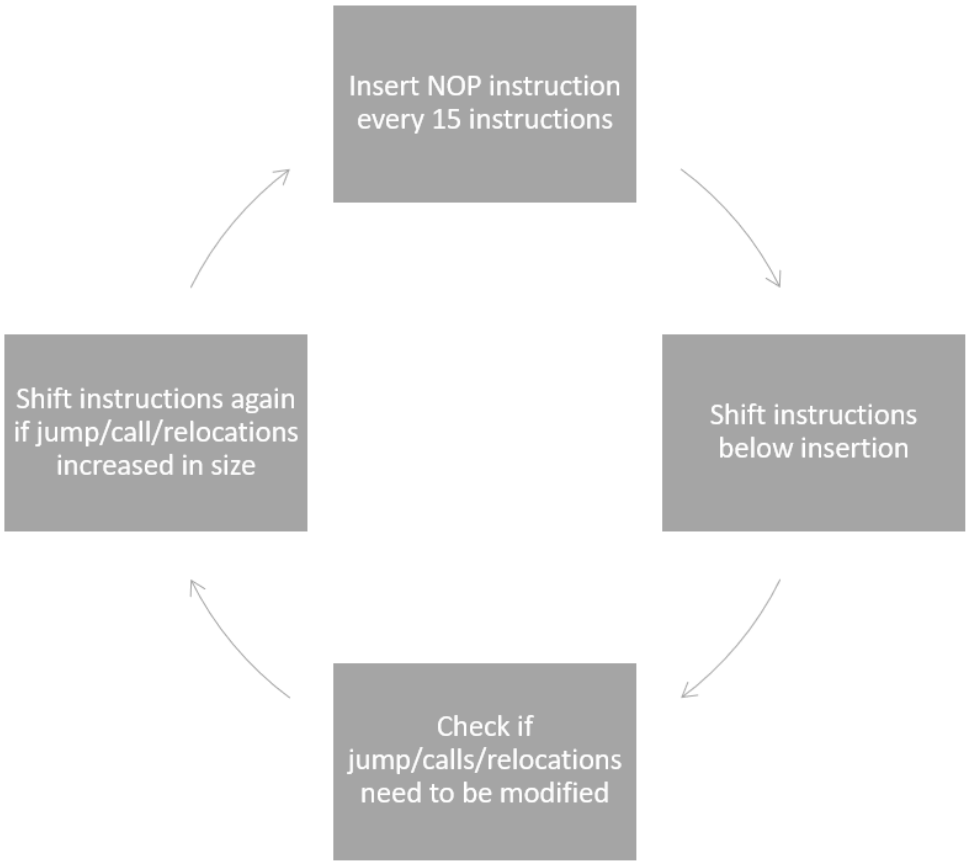


Figure 6. Insertion process



The insertion process begins with the insertion of a NOP instruction every fifteen instructions. Fifteen was the chosen amount to balance successful patches while still showcasing patcher strength. When the count was lowered, less binaries were able to be patched. If the count was increased, more binaries were able to be patched, but less insertions were made. If fewer insertions are made, the creditability of the patcher is reduced. So, fifteen was chosen as it was still a relatively small number and had a decent patch rate. Any instruction below this in the same section will be shifted by one byte to compensate for the insertion. Additionally, the jumps, calls, and relocations must be checked to see if any modifications to them need to be made. If there are modifications and those modifications

cause the control flow instructions to increase in size, then the instructions must be shifted down further. This process repeats until all the NOPs have been inserted.

The second design cycle starts with using the first artifact as the base with the insertion of NOPs being removed from its functionality. The second artifact will first gather all the ROP gadgets for the target binary. After this, it will check the address of the ROP gadget and see if it lands in the middle of an instruction. If it does, the address will be decreased by one until the address is correlated to the instruction that is causing it. This is done until all the ROP generating instructions have been marked. After this, each instruction will have a list of instructions created that will remove the ROP gadget. These lists of instructions will be inserted directly below each ROP generating instruction using the functionality of the first artifact. Additionally, the original ROP generating instruction will be replaced with NOPs if needed. Once all the insertions have been completed, the binary will be rebuilt using artifact #1's capabilities. This whole process will need to be done multiple times due to the fact that new ROP generating instructions can be created with the shifting of instructions.

Population

The population for the research is Intel x64 ELF binaries. These were chosen because ELF has a more simplistic binary format than EXE binaries do. ELF has three header structures while EXE has eleven header structures that need to be tracked (Saleh, 2020). Also, while Linux is not as widespread as Windows, it still has enough reach to warrant research into. Intel is also extremely widely used and allows for variable length instructions. This variable length instruction set is what allows for the increase in ROP gadgets in binaries and thus was chosen as the language of choice.

Sample

Sampling in design research defines the connection between the sample and population and how the sample can be generalized to reflect the population as a whole (Cash et al., 2022). The sample chosen must reflect the chosen population to a great enough degree for the testing of the artifact to be considered valid.

The sample chosen for this study was the Coreutils package. This package was chosen for a variety of reasons. The first being that it should come innately with all Linux distributions (*Coreutils.Git - GNU Coreutils*, n.d.). This would mean that should the artifact work on this package, it should work on all Linux distributions. For sample size concerns, Coreutils contains over 100 binaries. In addition, the binaries are also Intel x64 ELF binaries. So, should the tool be able remove the ROP gadgets from these binaries, it should be able to be expanded to other Intel x64 ELF binaries. Intel x86 ELF binaries were not tested during this study. However, most of the variances between the two architectures involve register size. Since the same register sizes were supported for x64 binaries, there is a strong likelihood that x86 binaries should work as well.

Data Collection and Instrumentation

The tools used for data collection and instrumentation are either standard Linux utilities or tools used in other studies in similar fields. The data collected involves the size of the binaries, number of ROP gadgets, and performance change. Each of these metrics has their own list of tools needed to collect this data.

The environment that testing was performed in was a virtualized Linux environment. The environment was virtualized using the Windows Subsystem for Linux on a Windows 11

computer. The chosen Linux environment was Ubuntu version Ubuntu 20.04.6. The installed version of Coreutils was 8.30-3ubuntu2.

To handle instrumentation and data collection of the size of the binaries, the *du* command in Linux was used. This command returns the size of the file that the command is run against. This command was run before and after the static binary rewriter was run against the tool. This shows the impact the tool has on the size of the binary that is being rewritten.

To determine the number of ROP gadgets, the Ropper tool was used. This tool reads through the byte code of the binary and returns any potential ROP gadgets it finds. The number of ROP gadgets was recorded by using Ropper against the target binary. After that information was stored, the tool was run against the binary to try to eliminate any ROP gadgets that the tool was targeting. Ropper was then run against the binary again and the amount of ROP gadgets was recorded a second time.

For the measuring of time, another built-in Linux utility was used. The utility in question is the *time* utility. *Time* gives a detailed report about the execution time of the binary it is run with. Similar to the other data collection metrics, this utility was run against the target binary both before and after the tool was ran against it. This provides data for the before and after state of the binary after the tool has been run against it.

All these data points should be valid for a variety of reasons. First, the tools in question are either built directly into certain distributions of Linux and peer-reviewed or have been used in studies similar to mine (*Coreutils.Git - GNU Coreutils*, n.d.; Flores-Montoya & Schulte, 2020; sash, 2014; Wang et al., 2017) . Secondly, the collection of data happens before and directly after the tool has been used. This means that no outside sources should be

able to modify the file and the only item being measured should be the impact the tool is having on the binary.

Reliability and Validity

The reliability of the instruments used and the validity of the data collected must be certified. Reliability of the instruments ensures that metrics gathered will always be the same. Validity ensures that the data collected from the tools is accurate. Both reliability and validity are needed to guarantee proper research. Also, the validity of expanding the results from the sample to the rest of the population must be verified as well. To prove these factors, a look at the tools used and the data collected was performed.

The *du* and *time* commands were designed for all Linux distributions. This means that the results from the *du* command and the *time* command should be the same no matter what Linux distribution it was used on. Additionally, these tools were created to perform specific tasks. The *du* command was created to determine the size of the files on Linux systems. The *time* command was created to measure the time a binary takes to run from start of execution to finish. Both commands measure metrics that were being studied and were native to Linux systems. Because of these reasons, these commands were chosen.

Capstone was used for at least three other reassemblable disassemblers (Flores-Montoya & Schulte, 2020; R. Wang et al., 2017; Wang et al., 2016). Since Capstone was used in peer-reviewed papers, there was a consensus on the reliability and validity of this tool. Keystone was a sister project to Capstone and is used in a variety of emulation platforms (Keystone, n.d.). The tool was presented at Blackhat, which is a large security conference in the U.S.A. (Keystone, n.d.). Since Keystone is used in a wide variety of emulation platforms and was approved to be presented at Blackhat, it should be able to be considered reliable and

valid. Ropper is a package that is supported by the Kali Linux distribution (*Ropper | Kali Linux Tools*, n.d.). It is also recommended for use as a ROP gadget finder. (*ROP Emporium*, n.d.). Since it is native on Kali Linux and is highly regarded, Ropper can be deemed a reliable and valid tool. So, Capstone, Keystone, and Ropper should be deemed reliable and valid tools and were chosen for this research because of that.

In terms of data, two areas need to be proven as valid. The first area is that the data collected is valid. The previous sections outlined how the tools used were both reliable and valid. This ensures that the data collected from them should also be reliable and valid. In addition, the only modifications that the binary undergoes are the modifications that the tool makes. This would mean that any data collected from the binary would come from either the tool or reliable instruments, thus meaning the data collected is directly related to the tool. The other area to address is expanding the sample's validity to the entire population. Firstly, the Intel x64 instruction set does not change from one binary to another. So, if the instructions can be changed, they should work for most binaries with similar compilations. Secondly, the ELF architecture also does not change from distribution to distribution. This would mean that if the sample can modify the binary, it should also work for all other ELF binaries.

This section addressed the concerns of reliability and validity of the study. The reasoning on why the tools used in this study are reliable and valid was given. Additionally, the methods taken to ensure that the data collected was solely related to the artifacts were shown. Also, methods of showing that the chosen sample set can be used to prove the population were examined. Because of this, this study can be marked as reliable and valid.

Data Analysis

The data for this study was analyzed by storing the information run from the tools and doing comparisons on them. The data gathered was collected both before and after the tool ran. The data before and after was needed to determine what effects the artifacts were having on the patched binaries. Only the artifacts modified the binaries, so any changes were directly related to the artifacts. The output of the statistics tools for the before and after results were stored in separate tables. The data analysis compared those tables to ensure that the results were what were to be expected as the output of the artifact.

Further information about the details of the population and the sample of the study was also provided. Additionally, information about the tools used for this study and why they were valid was also supplied. The validity of the data collected and the validity of expanding the sample to the entire population was verified.

The next chapter will present the findings and go into more technical detail about how the artifacts were created. It will also describe the limitations of each of the artifacts that were created during this study. Additionally, all the gathered metrics for each of the artifacts will be shown. Lastly, a summary of all the findings will be given along with the impact those findings have on the results of this research.

CHAPTER 5

RESULTS AND APPROACH

Introduction

This chapter will explain in more detail the technical aspects of the implementation of the artifacts. It will also highlight the findings of the artifacts in terms of success rate, binary size increase, and execution size. A conversation will be had about what these findings mean about the artifacts and the overall results. In addition, the limitations of the artifacts will be noted.

Overview of Artifacts

Both artifacts were created using Python with an object-oriented approach. Table 1 lists the names of the classes and gives a brief description of those classes. All the classes except for *Gamindustri* are used almost exclusively for data classification and storage for specific items. All data being used or stored is in relation to the binary that is being modified. For classes, the only difference between artifact #1 and artifact #2 will be in the *Gamindustri* class.

The key difference between artifact #1's *Gamindustri* and artifact #2's *Gamindustri* is the determination of where to insert instructions and what instructions to insert. Artifact #1 takes a simple approach of inserting a singular NOP below every fifteen instructions in the binary. In contrast, artifact #2 needs to dynamically discover the ROP generating instructions' addresses and create a list of instructions to remove the ROP generating instructions.

Table 1. Base Classes of the Artifacts

Class Name	Description
Elf_Header	Contains information about the ELF header
Section	Contains information about the sections
Segment	Contains information about the segments
Symtab	Contains information of the static symbol table
Dynamic	Contains information of the dynamic symbol table
Relocation	Contains information about the relocation table
GotPlt	Contains information about the Procedure Linkage Table of the Global Offset Table
RelaPlt	Contains information about the Procedure Linkage Table of the relocation table
Instruction	Contains information about each instruction
Gameindustri	Driver class for each artifact

Both of the artifacts have a variety of imports needed for functionality. Table 2 covers what those imports are and gives a brief description of why they are needed. The *ropper* and *operator* imports are solely for artifact #2.

Table 2. Imports of the Artifacts

Import Name	Description
elftools.elf.elffile	Allows for the loading and parsing of ELF files
elftools.elf.enums	Mappings of enum names to values
elftools.elf.relocation	Parses the relocation section of the ELF file
capstone	Disassembles instructions
keystone	Reassembles instructions
struct	Used for packing and unpacking byte arrays
re	Allows for the use of regular expressions
ropper	Finds ROP gadgets for a loaded binary
sys	Allows for the parsing of command line arguments
operator	Helps with sorting the gadget address from *ropper*

Each test for every binary in the Coreutils package varied based on what the binary was designed for. In other words, the *ls* binary is used to list directories in the system and thus was tested by using its various options in different directories. Also, the *nl* binary interacts with files and was fed multiple files for testing. All the binaries tested followed this approach. The results of these tests will be shown later in this chapter.

The following sections will highlight how the created *Gamindustri* driver class is used for both of the created artifacts. The steps the artifacts used to achieve their functionality will

be outlined. Also, the limitations and issues for each of the artifacts will be discussed. Lastly, the results of both the artifacts will be shown and analyzed.

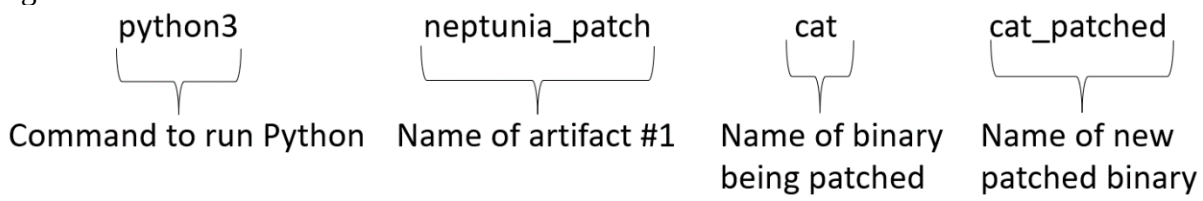
Artifact #1

The first artifact was designed to determine the feasibility of creating a static binary rewriter that did not rely on symbolizing during the patching process. To prove this, the inserting of NOPs into the binaries in Coreutils packages was performed. The following sections will outline the design process and decisions taken in creating artifact #1 in more technical detail. It will also outline the proper procedures on how the artifact is run and expected output. In addition, a description of how the testing was conducted will be noted. Also, current limitations with the artifact and how they impact the overall performance and results will be discussed. Lastly, the results of the current implementation will be shown.

Main Procedure and Running of Artifact #1

To run artifact #1, the name of the binary being patched and the name of the newly patched binary must be provided. An example is given in Figure 7. The execution of the artifact will halt if either the name of the binary being patched, or the new name for the patched binary was not supplied.

Figure 7. Command to run artifact #1



Once the artifact has been started, the *Gamindustri* class will ingest the name of the target binary and perform each step needed for determining ELF characteristics and instruction information for the binary. These steps will be listed in the upcoming sections. After this, the *create_insert_list* method of the *Gamindustri* instance will be called. This method will handle the insertion of NOPs every fifteen instructions. Lastly, the *build* method is called and will attempt to write the patched version of the binary with the name supplied in the command line argument. If everything was successful, the string "done" should print out in the terminal.

Determining ELF and Instruction Information

When the *Gamindustri* class is first initialized, information about the binary being patched must be parsed. The first step is to iterate through each segment of the binary and store them in a list of *Segment* class types. Additionally, if the flags of the segment were 5 or 2, they will also be stored in another list of *Segment* class types or a list of *Dynamic* class types respectively.

After the parsing of segments has been completed, the next step is to iterate over the sections. The printed name of each section will be checked for a variety of options and be stored in one or more lists of *Symtab, Relocation, GotPlt,* or *RelaPlt* class types. Additionally, if a section's flags are marked as executable, the instructions of that section will be gathered using the *get_instruction_list* method.

get_instruction_list loops through each section's data and converts the data to each instruction using Capstone. To determine when it has parsed every instruction, a counter starts at zero and is increased by the size of every instruction until the size matches the section's size. During each section pass, every Capstone instruction will be converted to an *Instruction*

object. Table 3 shows each of the values that the instruction will set when the object is created. Each instruction will also be passed into the *creator* method.

Table 3. Instruction Class Variables

Variable Name	Description
mnemonic	Stores the mnemonic of the instruction
op_str	Stores the operation string of the instruction
address	Stores the current address of the instruction
bytes	Stores the byte representation of the instruction
old_address	Stores the original address of the instruction
inserted	Keeps track of whether the instruction was inserted or was an original instruction
section	Stores the section that the instruction is found in
data	Check for if the instruction is data

The *creator* method is the function that will store each instruction passed into it in a list called *instructions*. This list will store every instruction in the binary in address order. The *creator* method also sets the section variable for each instruction to the proper section. After an instruction is added into the *instructions* list, the *creator* method will check to see if the instruction needs to be additionally stored in *jmp_list* or *call_list*, which store all the jumps and calls of the binary respectively. Lastly, the *creator* method will pass the instruction into the *get_additional_inst_info* method.

The purpose of the *get_additional_inst_info* method is to check if the instruction passed in matches specific types of instructions. If it matches one of them, extra information will be attached to the instruction and used when it is added to the *instructions* list in *creator.* Additionally, all passed in instructions will be stored in the *inst_dict* for use later in the artifact. The specific types of instructions are RIP offset instructions and jump instructions. For the RIP offset instructions, the target address is calculated by adding the instruction's address, size, and offset amount. If the target address is not located in the .text section, the address of the target address and RIP offset instruction will be stored in the *relatives* list. If the target address is in the .text section, the RIP offset instruction will be stored in the *text_relatives* list. As for the jump related instructions, the size of the jump instruction is calculated to determine what bytes of the instruction are related to the offset. The bytes can then be used to calculate the offset. Once the offset is determined, the address of the instruction, size of the instruction, and the offset will be used to calculate the target jump location. This target location and RIP offset will be stored in the current *instruction* variable.

After all these functions are finished, all information needed to perform the insertion of instructions should be gathered. This includes storing general ELF information, sections and their instructions, and any additional instruction information related to changing control flow. The next step artifact #1 performs is the insertion of NOPs into the binary.

Insertion of NOPs

With all the needed binary information gathered and stored, the insertion of NOPs into the binary can begin. This will involve cycling through the binary and finding every fifteenth instruction and storing those in a list called *bad_insts*. Then, a NOP will be inserted below each of these fifteenth instructions. Each of these insertions will also check if any control flow

instruction, such as jumps and calls, will need to be shifted or updated because of the size increase of the binary. To perform these operations, the *create_insert_list* method will be called.

The *create_insert_list* method works by cycling through each of the instructions in the *instructions* list and storing each of the fifteenth instructions into a separate *bad_insts* list. After the *instructions* list has been fully traversed, each instruction and instruction address in the *bad_insts* list will be passed to the *create_insertion* method one at a time to further the insertion process.

The *create_insertion* method uses Keystone to generate the op code for the NOP instruction. The address passed into the function, along with the Keystone op code, will be used to create an *Instruction* object. The created NOP's address will be at the same address as the passed in address after creation. To account for this, the address will be shifted by the passed in instruction's size to insert the NOP directly below it. The next series of methods will be dedicated to shifting instructions based on the insertion and the modifying of control flow instructions.

The following three methods involve the shifting of basic instructions and control flow instructions within the binary: *insert_and_shift, jmp_conversion_shift,* and *create_jmp_conversion. insert_and_shift* is the main driver of the three methods. The first task of *insert_and_shift is* to cycle through each instruction in the *instruction* list and mark the index of where in the list the insertion address resides. Once marked, each instruction with a higher index in the list will have its address shifted by the insertion amount. The instruction being inserted will then be added to the *instruction* list at the marked offset. The next step is to handle the shifting of the jumps within the binary.

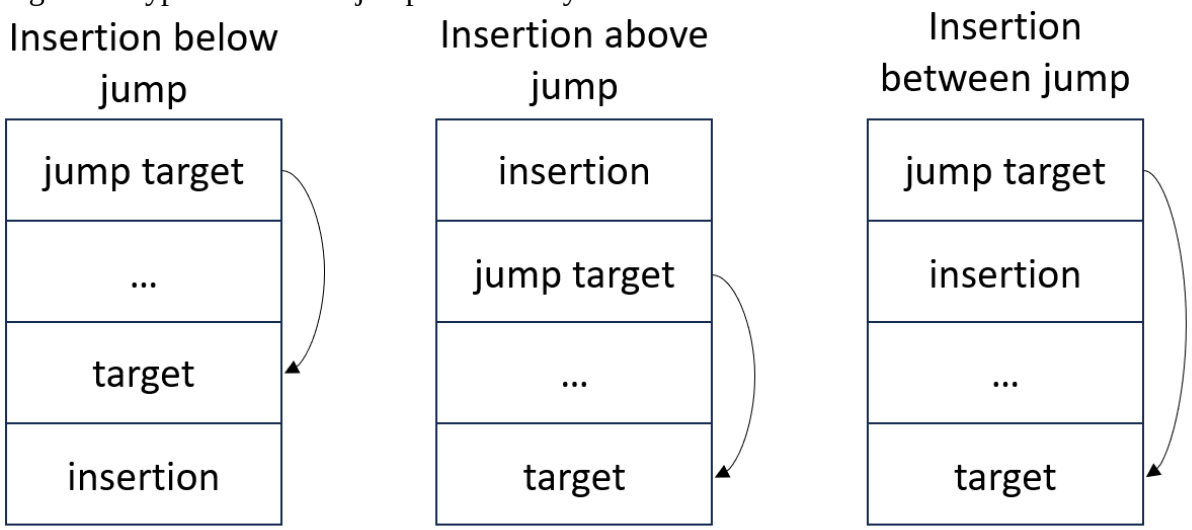
Before the shifting of the jumps can occur, there are multiple variations of jumps that need to be taken into consideration. The first of these variations is whether the offset of the jump is positive or negative. This study will refer to positive offsets as forward jumps and negative offsets as backwards jumps. There are six variations of these forwards and backwards jumps in relation to the inserted instruction that are possible in a binary. Figure 8 shows the variations for a forward jump. The backwards jumps are similar, except the jump target and jump instruction will be flipped. In addition to the insertion location, the other consideration that must be made is the size of the jump instruction. In most cases, the size of a jump instruction will be two or five bytes. However, the size of a jump can change from two to five bytes based on the size of the offset. All these variations of jumps will be handled differently by the *insertion_and_shift* method.

To handle the shifting of jumps, a pass will be made through the *jmp_list* list. During this pass, a variety of actions for each jump will be taken. First, which type of the six jumps the jump belongs to will be determined. Next, each jump instruction will have either or both of its *jmp_addr* or *rip_offset* modified depending on what type of jump it is. Additionally, if it is one of the types of jumps that will increase or lower the *rip_offset* size, the *jmp_conversion_shift* method will be called to check to see if the size of the jump needs to change. This method will be covered in a later paragraph. The *insert_and_shift* method keeps a running total of the insertion size of the original instruction and the size increase added from jumps converting from two bytes to five bytes. After this initial pass is done, a continuous pass over the *jmp_list* list will be made until no jump conversions are needed.

Once the continuous passes are done, the instructions in *call_list* and *text_relatives* will be shifted in a similar fashion to the jumps. However, both *call_list* and *text_relatives*

will use the total size of the insertion plus the increase of converted jumps to determine the amount they will be shifted by. Also, since they should not have variable length versions, there should be no need to check for size conversions. Once all this is done, the last step is to rebuild the binary.

Figure 8. Types of forward jumps in a binary



The *jmp_conversion_shift* method will determine if the instructions in a binary need to be shifted by an additional amount if a jump instruction needs to be increased in size. To accomplish this, it will analyze each jump instruction's *size* and *rip_offset* during the passes of the *jmp_list* list in *insert_and_shift*. If the size is two and the *rip_offset* is greater than 127 or less than -128, the jump will need to be converted to a five-byte jump instruction. This will require a multi-step process. This process will involve using the *create_jmp_conversion* method, which uses Keystone and the current jump instruction's values to create a new *Instruction* object. The five steps for this process are shown below:

1. Find the index after the current jump instruction.

2. Create a new jump instruction with the *create_jmp_conversion* method.
3. Replace the old jump instruction's *mnemonic* and *bytes* with NOPs.
4. Remove the old jump instruction from *jmp_list* and replace it with the new one.
5. Add the new jump instruction to *instructions* at the index in step 1.
6. Make a pass over *jmp_list, call_list,* and *text_relatives* and shift them if needed from the size increase.

Rebuilding of Binary

The first step of rebuilding the binary is checking if any of the entries in some of the binary tables, such as the global offset table or symbol table, need to be shifted. This is done by checking if an *instruction.old_address* is contained in the structure. If one is found, that relocation will be converted to the new instruction. Only instructions that were not added will need to be checked for these modifications.

After an initial shift of all the binary tables, the second step is to check the size of the text segment. If the size of the text segment exceeds its built-in padding size, it will need to be shifted by an offset amount of some factor of 0x1000. This increase will require each *segment's offset, virtual_address,* and *physical address* to be shifted by that amount. Additionally, each *section* will need to have its *offset* and *address* increased by that same amount. Lastly, each of the previous binary tables will need to have their entries shifted to accommodate the shift of locations of the instructions.

The third step is to fix the entry point of the binary and the bounds of the sections. The *header.entry* will be updated with the value in *entry.address.* This allows for the binary to know where the new entry point will be after all the shifting occurs. Next, a pass will be made

through the *instructions* list and the first and last instruction will be found for every given section. The *section.size, section.offset, and section.address* will be updated using the first and last instructions that were found.

The next step is to handle further shifting of the *relatives* list and an experimental fix to jump tables. The shift for *relatives* checks to see if the section the relative belongs to had its offset shifted. If it was, the relocation amount will need to be increased by the offset shift amount. To attempt to fix the jump tables, there is a multistep process:

1. Find a relocation in relatives that has a lea mnemonic.
2. Check if the relocation is in a read-only section.
3. Do a backwards search of a maximum of 20 instructions to look for the first found jump instruction.
4. Do another backwards search of 20 instructions starting at the found jump looking for an add instruction with the same operation string as the jump instruction.
5. Do one final backwards search from the add instruction for a lea instruction with a RIP register in the instruction.
6. Ensure that the instruction at the index of the last found instruction matches the relocation in step one.
7. Create a new *Jump_Table* object based on the previous steps information.

One final pass will be made through the tables in the binary to ensure that none of the previous modifications impacted them. After this, each *segment* and *section* will be packed and stored. Additionally, a byte string will continuously have *segment, relocation,* and *section*

information appended to it. Lastly, an attempt to add the information from the jump tables to the byte string will be made. This byte string will be used to rebuild the ELF binary.

Limitations

The limitations of the first artifact can be broken down into three different categories. The three categories are architectural limitations, symbolization limitations, and rebuilding limitations. Regarding architectural limitations, the artifact will only work for ELF x64 binaries. It should be able to be converted to ELF x86 binaries with some modifications, but Windows based binaries will not work. The issues with symbolization limitations mainly involve jump table or indirect jump concerns. For jump tables, sometimes the triple backwards search will add entries that are not actually jump tables. This will cause relocations to be incorrect and the patching process will halt. Indirect jumps, such as *notrack* jumps, were not grabbed because they use a register value as the offset. With the current implementation of artifact #1, there is no way to guarantee what that register value should be and, if one is encountered during execution of the patched binary, a segmentation fault is most likely. The binaries patched by the artifact, if there is no segment shift that occurred, will normally be seven bytes smaller than the original binary. This is caused during the *build* method when the segments are being aligned. This alignment seems to be misaligned in the last LOAD program header. This misalignment will strip, on average, seven bytes of padding from this LOAD program header. This should not be a large issue unless the padding in it is already completely used and actual data bytes are removed. All these limitations lead to a reduced accuracy of correctly patched programs. The limitations will also bleed into the effectiveness of the second artifact because it is built on top of the first artifact.

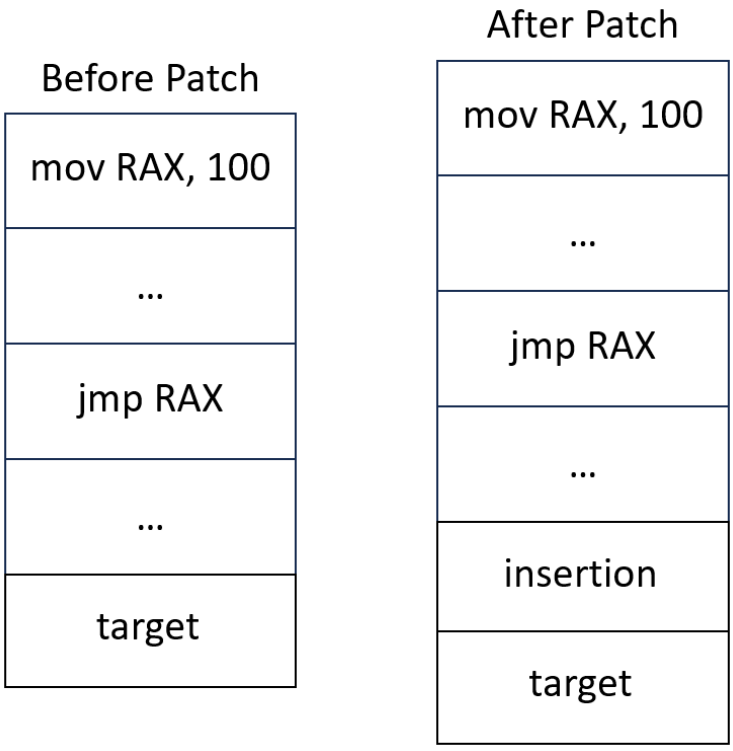
Findings

This section will detail the success rate of patching the Coreutils binaries and the metrics gathered from each successful patch. Table 4 lists all the successful patched binaries and the metrics for each of them. A binary was marked as successful if the default execution of the binary and multiple options ran without any errors. For example, if the *ls* binary worked in multiple directories and was able to use the -a, -l, and -al flags, it would be marked as successful. Each binary had separate test cases based on the utility provided by the binary. Once all the successfully patched binaries had been gathered, the following metrics were grabbed from them: runtime of the unpatched binary in hundreds of a second, runtime of the binary after patching in hundreds of a second, original size of the binary in bytes, and size of the binary after patching in bytes. The runtime was gathered by using the *time* command on each binary 100 times both before and after patching. The average of these 100 runs was used to calculate the run time. The size of the binaries was gathered using the *du -b* command to get the size of the binary in bytes.

Approximately 43% of the Coreutils binaries were able to be patched with artifact #1. The ones that were unable to be patched failed during the reassembly of the binary or failed during the runtime of the patched binary. Most of the failures happened during the experimental fix for the jump tables. The binaries that failed during execution were mostly related to the control flow instructions that were not targeted by the artifact, such as jmp <reg>. This issue can be shown by analyzing Figure 9. In Figure 9, the assumed distance of the jump is 100 away based on the value loaded in RAX. However, if an insertion is made in between the jump and the target, the distance will be incorrect. This can lead to control resuming at odd locations, which could lead to runtime errors. Both the jump table fixing fails

and control flow instruction issues were known limitations and will be addressed in future modifications to the artifact.

Figure 9. Example Jump Issue



While no testing was performed on binaries outside of the Coreutils binaries, any ELF x64 binary of similar sizes should be expected to produce similar results. There are two possible outliers in which differing results could be expected. The first is if a binary has a large number of indirect jumps. Since these types of jumps are not being tracked, the likelihood of an incorrect rebuild would be increased. The second case would be large binaries such as the Chrome browser. These large binaries require more inserts and thus there is a higher chance of a problematic insert. However, neither of these types of binaries were tested so the impact these would have is unclear.

The runtime of the binaries was not consistently faster or slower after being patched. This was due to the *time* command logging the overall runtime of the binary, which means that each binary will be affected by the overall Linux system. Each binary was run in the same state in the virtualized Ubuntu environment. This was done to attempt to limit these differences. However, with the differences being minor and with the patched runtime sometimes being faster, the patch execution speed seems to have a very minor effect and the system has more of an effect on the runtime of the artifact.

The patched size of the binary normally stayed the same, except for the seven bytes being removed as stated in the limitation's section, after the patch. This was due to each segment normally having enough padding, so they did not need to be shifted. For the ones that did need to be shifted, the size increase was around 4000 bytes.

Table 4. Test results for artifact #1

Binary Name	Original Runtime	Patched Runtime	Original Size	Patched Size
b2sum	2.18	2.3	59768	63857
base32	2.14	1.93	43352	43345
base64	2.1	1.94	43352	43345
basename	1.21	1.45	39256	39249
cat	2.3	2.17	43416	43409
cksum	1.89	2.05	39256	39249
dircolors	1.39	1.61	47456	47449
echo	1.5	1.21	39256	39249
env	1.6	1.55	43352	43345
expand	2.3	2.06	43384	43377
expr	1.49	1.73	55640	55633

factor	1.9	1.88	80248	84337
false	1.24	1.24	39256	39249
fmt	2.16	1.92	47448	47441
fold	2.06	1.94	43352	43345
groups	1.32	1.35	39256	39249
hostid	1.55	1.44	39256	39249
id	1.71	1.74	47480	47473
link	1.52	1.7	39256	39249

Table 4, continued.

Binary Name	**Original Runtime**	**Patched Runtime**	**Original Size**	**Patched Size**
ln	2.4	2.34	76160	76153
logname	1.15	1.13	39256	39249
mknod	1.58	1.78	72024	76113
nice	1.27	1.28	43352	43345
nl	2.13	1.87	43448	43441
nproc	1.52	1.3	43352	43345
numfmt	1.66	1.71	67992	67985
paste	2.04	2.15	43384	43377
pathchk	1.41	1.33	39256	39249
pinky	1.39	1.46	43384	43377
printenv	1.26	1.2	39256	39249
realpath	1.76	1.61	51576	51569
shred	1.7	1.92	63864	67953
split	2.79	2.92	60184	60181
sync	1.93	1.97	39256	39249
tail	2.21	2.46	72088	76177
tee	1.41	1.49	43384	43377

true	1.31	1.27	39256	39249
unexpand	1.9	1.98	43384	43377
uniq	2.08	1.95	51576	51569
unlink	1.26	1.11	39256	39249
uptime	1.11	1.1	14568	14561
wc	2.19	2.22	47456	47449
yes	1.37	1.37	39256	39249

Artifact #2

The goal of artifact #2 was to build upon artifact #1 to eliminate ROP gadget causing instructions in binaries. The modifications made to artifact #1 are found in the process of determining the insertion list, what instructions to insert, and adding the ability to choose certain types of instructions to fix. Like artifact #1, the following sections will outline the design process and decisions taken in creating artifact #2 in more technical detail. It will also outline the proper procedures on how the artifact is run and expected output. In addition, a description of how the testing was conducted will be noted. Also, current limitations with the artifact and how they impact the overall performance and results will be discussed. Lastly, the results of the current implementation will be shown.

Main Procedure and Running of Artifact #2

To run artifact #2, the name of the binary being patched and the name of the newly patched binary must be provided. An example is provided in Figure 10. The execution of the artifact will halt if either the name of the binary being patched or the new name for the patched binary was not supplied. Artifact #2 also supports a list of options shown in table 5 that can be supplied to the artifact.

Figure 10. Command to run artifact #2

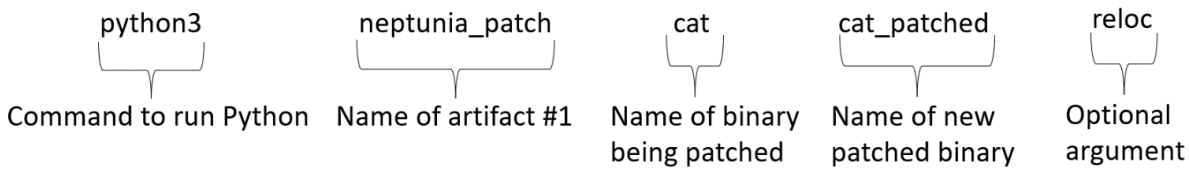


Table 5. Options for Artifact #2

Option Name	Description
check	Do not patch and simply display gadget list and gadget count information
reloc	Allows for patching of certain relocation instructions
call	Allows for patching of certain call instructions
jmp	Allows for patching of certain jump instructions
all	Enable the reloc, call, and jmp options

The overall execution structure of artifact #2 is the same as artifact #1 with some added analysis and parsing. The first addition is the setup of integrating *Ropper* into the binary. This is done by calling *RopperService* and configuring it to target the binary being patched and setting it to find only ROP gadgets for the x86-64 architecture. These found ROP gadgets will be used in the *create_insert_list* method to determine what instructions to modify rather than just grabbing every fifteenth instruction. The second addition is a new method called *get_instruction_information*. This method will take a given instruction and will return information about the instruction, such as the register sizes being used, for use when trying to replace ROP generating instructions. The third addition is the *fix_rop_instruction* method. This method will have multiple sections dedicated to it explaining what ROP generating instructions it can fix and how it fixes them. The last addition is the printing of ROP information of the binary being patched. This information entails the total amount of ROP gadgets, list of potential ROP generating instructions, and total count of potential ROP generating instructions.

Other than all these additions, artifact #2 will run in the same way as artifact #1. An ingested binary will be selected for patching. The needed information will be parsed and stored in a *Gamindustri* object. After this, the *create_insert_list* method will still determine the instruction locations that need to be modified and handle calling the needed methods to fix them. The *build* method will also still handle writing all the changes to a new binary.

Determining ROP Generating Instructions

When using *Ropper* with the options set to 'rop', the returned gadgets will not always be a standard ret gadget. To determine which gadgets to look for, a pass will be made through each gadget *Ropper* found and check if they contain the "ret" mnemonic in them. If they do contain the mnemonic, the address of the gadget, which is supplied by *Ropper*, will be checked against the *inst_dict* to see if that address is in the dictionary. If the address is in the dictionary, it means that it is an actual ret instruction and can be ignored. So, if it is not in the dictionary, that means the "c3" byte is in the middle of an instruction. To determine the instruction that contains this byte, the address of the gadget will be subtracted by one until the current address location matches one in *inst_dict.* The found instruction will then be added to the *bad_ints* list. Once every gadget from *Ropper* has been parsed, each instruction in *bad_insts* will be passed to *fix_rop_instruction* to attempt to remove the "c3" byte from the instruction.

Classification of ROP Generating Instructions

The first step of the *fix_rop_instruction* method is to determine what type of ROP generating instruction it is. The initial classification can be broken into two types:

1. Instruction's operation string contains a "c3" in it.
2. The bytes that make up the instruction contain the "c3".

These two types are also split into sub-types when trying to be fixed. The first type is currently only broken down into one sub-type:

1. Instruction contains a memory operation with the RIP register.

The second type is broken down into three sub-types:

1. Instruction is a register-to-register or register-to-constant operation.
2. Instruction mnemonic contains a jump.
3. Instruction mnemonic contains a call.

Each of these types are solved using different approaches. They will all be given their own sections describing how their approaches differ.

Fixing Memory Operation Instructions

For this study, if an assembly instruction involves the use of []'s in its operation string, it will be classified as a memory operation instruction. Currently, only a small subset of these memory operation instructions is supported. To be marked as a part of the supported subset, there are three requirements the instruction must meet:

1. The "c3" byte must be in the operation string.
2. The instruction mnemonic must be lea or mov.
3. The memory access must involve the use of the RIP register.

If any of these requirements are not met, the instruction will not be selected for patching.

Once a memory operation instruction has been selected for patching, a variety of operations will be performed to remove the bad bytes. The first step is to determine what bytes of the offset have a "c3" in them. This can be done by looping through the last four bytes of the instruction and checking to see if any of them equal "c3". If a byte does equal "c3", a *fix* variable will have its value increased by 256 raised to a power of the current loop

iteration. In other words, if the first byte is “c3”, *fix* will be increased by one and if the second byte is “c3”, *fix* will be increased by an additional 256. This will result in a value that, when subtracted from the original offset, will remove any instances of the “c3” byte from the offset. The *new* variable is set by taking the original offset, subtracting *fix* from it, and subtracting the instruction size from it. The instruction size gets subtracted due to the original instruction remaining, which increases the RIP value. With these values, the instructions listed in table 6 can be inserted below the original instruction and the original instruction can be NOPed out.

Table 6. Memory operation instruction modifications

Original Instruction	**New Instructions**
lea <reg>, [rip +/- <offset>]	lea <reg>, [rip +/- <new>] lea <reg>, [<reg> + <fix>]
mov <reg>, [rip +/- <offset>]	lea <reg>, [rip +/- <new>] mov <reg>, [<reg> + <fix>]

One additional concern when modifying these instructions is the integrity of the relocations during the rebuild process. To handle this, when a memory operation is being patched, it will be removed from the *relatives* list. However, the original target of the relocation will be saved. This is because when the first new instruction is inserted, it will be added to the *relatives* list with the original offset as its target value. To account for this, the insert into the *relatives* list will also contain the *fix* value. The *fix* value will be used to reduce the offset value when checks are made against the target address. This ensures that the same target address is valid even though the offset value will not align.

Fixing General Instructions

General instructions are the largest set of instructions that get removed from the binary. In this study, they are defined as an assembly instruction that performs a register-to-register operation or register-to-constant operation. There are some more limitations for the current implementation of the tool. These limitations are as follows:

1. The mnemonic must be mov, add, or sub.
2. The first register must be a b or r11 register.
3. If the second operand is a register, it must be either an a or r8 register.
4. If the second operand is a constant, the mnemonic must be either mov or add.

With these limitations, it can be assured that a c register is not in use during the instruction. This means that the c register can always be used to fix the instruction. The use of the b or r8 registers with specific operands causes the “c3” byte to be generated. Because of this, a c register can be used to remove the register or operand causing the bad byte to be generated. To do this, the selected value will be exchanged with the c register, use the c register in the original instruction, and then re-exchange the c register with its original value. Table 7 showcases how this is done.

Table 7. General instruction modifications

Original Instruction	New Instructions
<mnemonic> <b_reg>, <a_reg>	xchg <c_reg>, <a_reg> <mnemonic> <b_reg>, <c_reg> xchg <c_reg>, <a_reg>
<mnemonic> <b_reg>, <r8_reg>	xchg <c_reg>, <r8_reg> <mnemonic> <b_reg>, <c_reg> xchg <c_reg>, <r8_reg>
<mnemonic> <b_reg>, <constant>	xchg <c_reg>, <b_reg> <mnemonic> <c_reg>, <constant> xchg <c_reg>, <b_reg>
<mnemonic> <r11_reg>, <a_reg>	xchg <c_reg>, <a_reg> <mnemonic> <r11_reg>, <c_reg> xchg <c_reg>, <a_reg>
<mnemonic> <r11_reg>, <r8_reg>	xchg <c_reg>, <r8_reg> <mnemonic> <r11_reg>, <c_reg> xchg <c_reg>, <r8_reg>
<mnemonic> <r11_reg>, <constant>	xchg <c_reg>, <r11_reg> <mnemonic> <c_reg>, <constant> xchg <c_reg>, <r11_reg>

The xchg instruction does not modify any flags during execution. This means that exchanging with the c register will not affect the integrity of the original program. Also, the c register is swapped back to its original value right after the newly added instruction. So, the contents of the register will be preserved and will be restored for use later during program execution.

Fixing Jump Instructions

The targeted jump instructions are a subset of the types of jumps that x64 permits. The only jumps that are targeted are what this study will refer to as standard jumps. These jumps include jmp, jnz, jge, etc… They do not include more advanced jumps such as notrack jmp or jrcxz. Additionally, the jump must not use a register or memory location to determine the target. The target must be located only using an offset. There are two methods to fix these standard jumps. The first method will insert a NOP above or below depending on if it is a forward or backwards jump. The second method will insert two additional jumps into the binary.

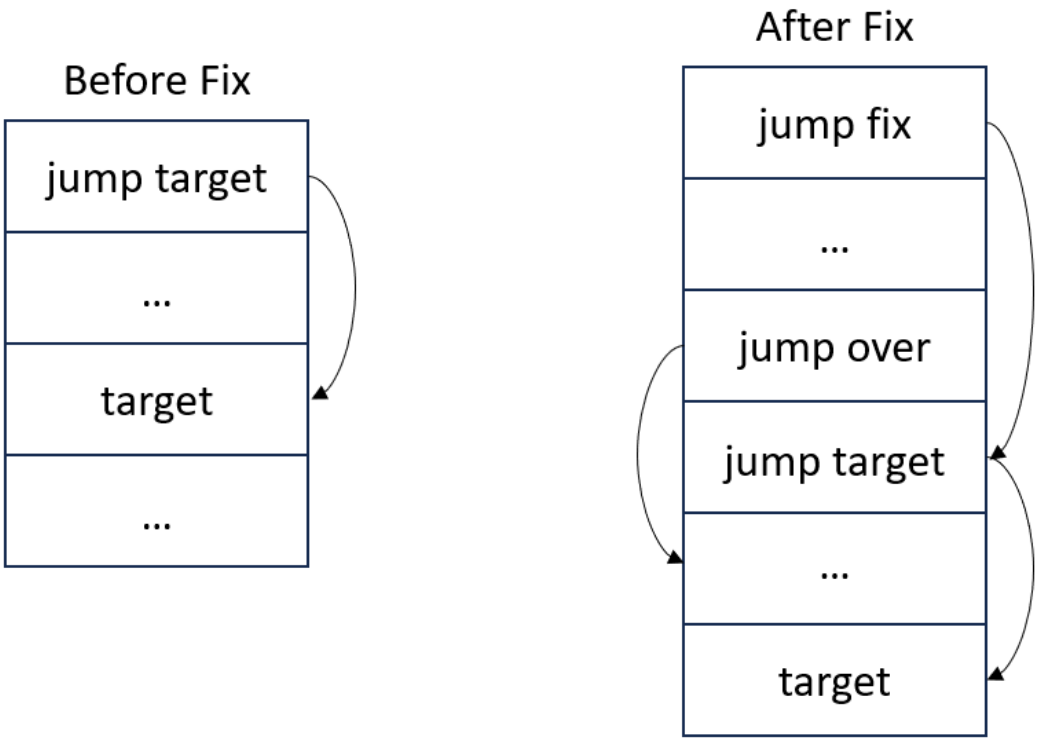
To determine which method to use, the bytes of the instruction related to the offset amount will be checked for the "c3" byte. If the bytes contain a "c3" only in the first byte, the first method will be used to fix the instruction. If any of the other bytes in the offset bytes contain a "c3" in them, the second method will be used to remove the "c3" from the instruction.

The first method works by first checking if the jump instruction is a forward or backwards jump. If the instruction is a forward jump, a NOP instruction will be inserted below the original jump instruction. Inversely, if the instruction is a backwards jump, a NOP instruction will be inserted above the original jump instruction. Since both instructions will be modified by the *insert_and_shift* method, the offset of the jump instruction will be increased by one. By doing this, the offset of the jump instruction will no longer contain a "c3" in it.

The second method, in addition to determining if it is a forward or backwards jump, also needs to determine a new offset amount in the same way the fixing relocation instructions works. This new offset amount will also be referred to as *fix*. Once *fix* is calculated, the offset of the original instruction's offset will be modified by the *fix*'s value. The new offset will be

checked to ensure that it does not land in the middle of an instruction. If it does, the offset and *fix* will be modified one byte at a time until it lands at the beginning of an instruction. A jump will be inserted at this location to jump over a jump that will be inserted directly below this newly added jump. The jump added directly below will have an offset amount of *fix*'s value. Figure 11 showcases the before and after of this process. Both jumps will also be added to *jmp_list* for further modifications if needed.

Figure 11. Method two of fixing jump instructions



Fixing Call Instructions

For this study, instructions that contain a call mnemonic and contain an offset value in the operation string will be included in the patched call instructions. Any call instruction that uses a register or memory location to determine the callee will be excluded from the list of patched instructions. All calls will be fixed by inserting a NOP above or below the instruction

depending on the location of the callee's location. If the callee is at a lower address than the call instruction's location, then the NOP will be inserted above the call instruction. Vice versa for callees at a higher address.

Currently, only call instructions with the first byte of the offset bytes being a "c3" will be patched. This is because to fix the remaining bytes they would have to be increased by a power of 256 to shift the byte to a different value. So, to fix a byte at the 4^{th} slot of the offset bytes, 16,777,216 NOP instructions would need to be inserted to fix the offset byte. This was deemed not effective, and those call instructions were ignored.

Limitations

Artifact #2 shares the same limitations regarding the patching of the binary that artifact #1 has. However, artifact #2 also has some limitations for what type of instructions it targets. First, artifact #2 only targets ROP gadgets that are created from the "c3" byte. Second, only instructions that contain the "c3" in the middle of the instruction will be targeted for patching. Standard return statements will not be chosen for patching because of this. Third, only the subset of instructions for relocation, general, jump, and call that were mentioned in previous sections will be targeted by artifact #2. This means that some ROP generating instructions will remain with the current implementation of artifact #2. Lastly, some instructions that should be removed cannot be removed because of the limitations introduced from artifact #1.

Artifact #2 poses the chance of introducing additional ROP gadgets into the binary. This is due to the fact that the insertions to remove instructions containing the "c3" byte shifts the binary. These shifts can cause other gadgets, such as one containing "c2" bytes, to appear. These are not targeted and thus will remain in the binary. Additionally, there could be a

concern with new vulnerabilities being introduced into the binary as well with the new insertions. However, since the instructions added are equivalent to the old instructions, there does not seem to be a direct indication that this would be a problem. Also, the jumps added are direct jumps and not indirect jumps. Since the jumps added are direct, the concerns of helping JOP based attacks is unlikely.

Findings

Artifact #2 used the same procedures as artifact #1 to determine the success rate for each Coreutils binary. Artifact #2 includes the same metrics as artifact #1 with some additional metrics involving ROP gadgets. The additional metrics are the total amount of ROP gadgets before patching, the total amount of mid-instruction “c3” gadgets before patching, the total amount of ROP gadgets after patching, and the total amount of mid-instruction “c3” gadgets after patching. Table 8 lists each successful binary and their metrics. Artifact #2 had a higher successful patching rate than artifact #1 with a ~63% patch rate.

The total ROP gadget count does not directly align with the amount of mid-instruction ROP gadgets removed. The misalignment is caused by the shifting of instructions introducing or removing ROP gadgets that artifact #2 does not target. Some of the binaries, such as the sha binaries, still contain a large number of mid-instruction ROP gadgets. The reason there are so many in these binaries is because they have a large number of general instructions that the current implementation of artifact #2 does not target.

The size of the binaries was largely unaffected by the removal of the mid-instruction ROP gadgets. Most of the binaries had their size decreased due to the padding being stripped because of artifact #1’s build implementation. The size of some of binaries had to be

increased due to the segments needing to be shifted. However, these results show that the tool had relatively no impact on the size of the binaries that were patched with artifact #2.

The runtimes of the binaries before and after patching with artifact #2 had larger variances than the runtimes from artifact #1. This is probably caused by the fact that more complex instructions are being inserted into the binary, rather than just NOP instructions. While the runtimes do have a larger variance, there are still a large number of binaries that have a faster runtime than the original binaries. So, similar to the binaries patched with artifact #1, it would seem that the system variables have more of an impact on the runtimes of the binary than the patching itself.

For most binaries, based on the results, almost all the mid-instruction ROP gadgets can be removed from the binary. In some cases, all of them could be removed. The initial patch was the one that removed the most ROP gadgets from the binary. Each subsequent patch tended to only remove a couple of ROP gadgets. To achieve the exact same results, the user would need to use the same flags on each pass. However, similar results will be achieved by running multiple passes until the user is satisfied with the results.

Table 8. Test Results for Artifact #2

Binary Name	**Original Runtime**	**Patched Runtime**	**Original Size**	**Patched Size**	**Total ROP**	**Split ROP**	**New Total ROP**	**New Split ROP**
b2sum	2.18	2.38	59768	63857	296	64	245	9
base32	2.14	1.81	43352	43345	169	35	143	7
base64	2.1	1.78	43352	43345	171	35	145	9
basename	1.21	1.11	39256	39249	125	26	103	5
cat	2.3	1.68	43416	43409	156	38	121	5
cksum	1.89	2.25	39256	39249	125	26	100	2
csplit	4.31	7.53	55672	55665	222	54	192	15

cut	2.02	3.62	47480	47473	177	41	144	9
dir	3.82	2.56	142144	142137	731	158	616	24
dircolors	1.39	1.21	47456	47449	175	36	143	6
dirname	1.39	1.05	39256	39249	124	24	102	2
echo	1.5	1.37	39256	39249	127	22	108	2
env	1.6	1.63	43352	43345	155	41	120	6
expand	2.3	2.22	43384	43377	180	39	146	9
expr	1.49	2.03	55640	55633	209	33	172	2
factor	1.9	3.41	80248	84337	328	81	269	24
false	1.24	2.46	39256	39249	120	25	95	1
fold	2.06	2.32	43352	43345	153	38	130	12
groups	1.32	1.14	39256	39249	136	29	118	7
hostid	1.55	1.34	39256	39249	119	22	99	2

Table 8, continued.

Binary Name	**Original Runtime**	**Patched Runtime**	**Original Size**	**Patched Size**	**Total ROP**	**Split ROP**	**New Total ROP**	**New Split ROP**
id	1.71	2.81	47480	47473	174	35	143	6
join	2.49	3.68	55672	55665	196	34	173	5
link	1.52	2.05	39256	39249	122	26	103	6
logname	1.15	1.2	39256	39249	120	23	101	2
ls	6.61	3.25	142144	142137	731	158	614	25
md5sum	4.82	2.93	47480	47473	216	36	186	7
mkfifo	4	3.61	67928	67921	322	51	277	8
mknod	1.58	2.91	72024	76113	355	60	310	12
mktemp	1.84	2.37	47448	47441	205	53	165	14
nice	1.27	2.4	43352	43345	131	30	114	11
nl	2.13	2.67	43448	43441	155	36	119	5
nohup	1.32	1.28	43352	43345	140	30	122	11
nproc	1.52	1.26	43352	43345	145	27	122	2
paste	2.04	2.19	43384	43377	142	34	120	10
pathchk	1.41	1.41	39256	39249	132	27	106	2
pinky	1.39	1.33	43384	43377	160	38	134	6

printenv	1.26	1.02	39256	39249	116	22	96	2
ptx	2.35	2.23	80280	80273	321	86	260	13
pwd	2.28	1.04	43352	43345	138	27	113	1
readlink	1.99	1.96	51544	51537	207	37	174	3
realpath	2.17	1.78	51576	51569	230	46	191	6
rm	3.5	3.32	72056	72049	335	55	279	5
sha1sum	2.4	2.22	51576	51569	249	60	221	30
sha384sum	2.37	2.88	67960	67953	326	118	267	57
sha512sum	4.57	5.31	67960	67953	326	118	267	57
shuf	2.29	2.96	59736	59729	301	69	242	8
split	2.79	3.5	60184	60181	258	67	222	28
sum	2.84	4.08	47456	47449	185	45	147	7
sync	1.93	3.98	39256	39249	126	23	104	2

Table 8, continued.

Binary Name	**Original Runtime**	**Patched Runtime**	**Original Size**	**Patched Size**	**Total ROP**	**Split ROP**	**New Total ROP**	**New Split ROP**
tac	3.84	3.94	43352	43345	160	35	132	9
tee	1.41	1.75	43384	43377	147	40	123	11
timeout	3.38	2.7	43800	43797	164	34	133	3
true	1.31	2.01	39256	39249	119	24	94	1
truncate	1.85	1.98	43352	43345	134	28	110	4
tsort	4.14	3.16	43352	43345	157	38	133	10
tty	3.55	2.21	39256	39249	118	24	99	2
unexpand	1.74	2.54	43384	43377	173	27	148	6
uniq	2.08	2.15	51576	51569	190	41	162	13
unlink	1.26	1.75	39256	39249	190	41	162	13
uptime	1.11	1.58	14568	14561	17	3	15	0
users	1.01	1.2	39256	39249	126	26	104	3
vdir	8.04	8.22	142144	142137	731	158	616	23
wc	2.19	2.53	47456	47449	188	41	153	9
whoami	1.29	1.45	39256	39249	121	23	98	2

yes	1.37	1.09	39256	39249	122	25	98	1

Summary of Findings

Artifact #1 had a successful patch rate of ~43% of the Coreutils binaries while artifact #2 had a ~63% successful patch rate. To be marked as successful in this study, the binary had to be patched with no errors, the base binary had to run without runtime errors, and multiple flags (if the binary had them) had to work without runtime errors. Most of the errors during patching time came from the experimental jump table fixing in the *build* method. The runtime errors both for the base option and flags were normally caused when a register was used as the offset value for either a relocation or a jump. These types of instructions are not tracked with the current implementation of the artifacts.

The size of the binaries after being patched was normally reduced by a size of seven bytes. This held true for both artifact #1 and artifact #2. The cause of this reduction was the calculation for the alignment of the segments had an interaction with the last LOAD program header that caused some padding to be stripped. The size reduction should have no impact on the reliability of the program so long as the bytes being stripped are not data bytes. There were some cases where the size of the binary was increased. The increase came from the *build* method needing to shift some segments due to lack of sufficient padding to store all the additional instructions. The size increase was relatively small with it being ~4000 bytes.

The runtimes of the binaries were largely unaffected after being patched from both artifacts. Artifact #2 had a slightly larger variance and it is assumed to be caused by the complexity of the instructions being greater than just inserting NOPs. However, both artifacts had binaries that had faster runtimes than the original binary. Because of this, it seems that the

system variables have a greater impact on the runtime of the binary compared to being patched by both artifacts.

The number of mid-instruction ROP gadget instructions for almost all the binaries was greatly reduced after being patched multiple times by artifact #2. There were some binaries, such as the sha family of binaries, that still contained a large amount of mid-instruction ROP gadget instructions in them. This was caused by the specific instruction not being targeted by artifact #2. The largest decrease in the number of these ROP gadget causing instructions was during the first patch of artifact #2. There were diminishing returns after each subsequent pass and, in some cases, an increase of ROP gadgets. There is also a disconnect between the total ROP gadget count and the mid-instruction ROP gadget count. This is because the shifting of instructions can cause offsets to change and various other side effects.

In regard to other tools, artifact #1 was weaker than the other static binary rewriters. This is most likely because they gather more information about the binary than artifact #1. In terms of ROP removal, artifact #2 targeted a separate space than most other tools that attempted to mitigate ROP using static binary rewriting. Artifact #2 targeted mid-instruction ROP gadgets will other tools targeted generic return instructions. So, if the methods from artifact #2 can be combined with the methods from other tools most ROP gadgets could be neutralized.

CHAPTER 6

CONCLUSION

The purpose of this study was to determine the feasibility of using static binary rewriting to remove ROP gadgets from binaries. When this study was conducted, tools that focused on static binary rewriting attempted to symbolize binaries to perform their patching. The first artifact created by this study attempted to patch binaries without this symbolization. Additionally, at the time of this research, tools that attempted to remove ROP gadgets with static binary rewriting focused on the removal of standard return instructions with an emphasis on function returns. The second artifact created in this study focused on the removal of instructions that could cause ROP gadgets when their bytes were split.

Both artifacts were developed using DSR. A comparison was made against other methods of research for the chosen space of this research and DSR was the most suitable. One of the reasons was the flexibility that DSR allowed when designing the artifacts. Additionally, Hevner's guidelines and Wieringa methods were followed to provide a valid DSR approach.

Discussion of Findings

There are multiple areas to consider when analyzing the results of the research. These areas include successful patch rate, binary size, execution speed, and ROP gadget count. Successful patch rate was used as a metric to determine the accuracy of the artifacts and thus the overall usefulness of the artifact. Binary size was tracked to see if the artifacts inserting additional instructions into the binary created a large increase in size of the binary. Execution speed was tracked to see if the runtimes of the binary were increased beyond a reasonable level after patching was performed on the binaries. ROP gadget count was tracked to measure

and ensure that artifact #2 was able to remove ROP gadgets from the binary using the methods employed in this research. Each artifact had differing results in all these areas.

Based on the results from the study, artifact #1 does not perform well as an overall static binary rewriter. This is due to the limitations outlined previously about the inability to track specific control flow instructions. Additionally, some of the limitations cannot be solved using the simple backwards propagation techniques artifact #1 employs for patching jump tables. However, the basis of artifact #1 was useful in creating the more specialized static binary rewriter artifact #2. Artifact #2 had a 20% increase in patch rate in comparison to artifact #1. The increase was due to the more targeted approach for patching specific instructions rather than simply inserting a NOP every fifteen instructions. Patching specific instructions helped in three ways. First, the overall number of insertions was less than inserting the NOPs. Second, less instruction regions were affected by the insertions. Third, the targeted instructions were in areas that typically would not affect specific control structures. This means that, without further modifications to the artifacts, they should be used as specialized binary rewriters and not generic ones.

In most cases, neither of the artifacts had a large impact on the size of the binary. Except for a small number of cases, due to the alignment algorithm both artifacts employ, patching the binary caused the size of the binary to shrink. The padding of the segments allowed for new instructions to be inserted without needing to increase the size of the segments. In the few cases where they did need to be shifted, the size increase was only ~4000 bytes. This increase of size is relatively minor compared to the overall size of the binary. Additionally, the likelihood of a segment needing to be shifted again after receiving the additional 4096 bytes in padding is very low. This means that there would be a finite

amount that the size of the binary could be increased based on the insertions. So overall, patching using the artifacts had little to no effect on the size of the binaries.

Neither artifact had a large impact on the execution speed of the binaries after being patched. Artifact #1 only inserted NOPs, which have little overhead when being executed. Artifact #2 introduced more complex instructions at every insertion location but had fewer overall locations to insert instructions. Artifact #2 did have slightly more variability in execution speed, but it was not enough to warrant concern. Additionally, some binaries after being patched had faster execution times than their original counterparts. This means that the variabilities in the system had a greater impact on the runtime of the binaries than both artifacts. So, execution speed is not a concern when performing this type of static binary rewriting.

The last metric that was analyzed was ROP gadget count. Artifact #1 did not target ROP gadgets and thus the count was not analyzed for artifact #1. Artifact #2 targeted a subset of instructions that contained a “c3” byte in them. The subset of instructions was further broken down into different types. The types were memory operation instructions, general instructions, jump instructions, and call instructions. Each of these types had different ways of removing the original instruction that caused the ROP gadget while still maintaining original functionality of the binary. Artifact #2 was able to remove a vast majority of these mid-instruction ROP gadgets from most of the binaries in Coreutils. There were some outlier binaries, such as the sha family, that contained a large number of instructions that are not part of the targeted instructions for the current implementation of artifact #2.

Based on all these findings, all the metrics used for the study performed well. Artifact #2, which was the main point of this study, was able to achieve a ~63% patch rate, rarely

affected the size of the binary, had a minor impact on the execution speed of the binary, and was able to remove most, and in one case all, of the mid-instruction ROP gadgets. These favorable results showcase the feasibility of using static binary rewriting to remove instructions that generate ROP gadgets in binaries. So, static binary rewriting can be used to eliminate ROP gadgets from binaries.

Recommendations for Future Work

There are two main areas in which the methods could be improved. The first is improving the static binary rewriting methods. Currently, artifact #1 has limitations that could be improved on. The most pressing issue is the inability to track control flow instructions that use register values as their targets. This was because simple backwards propagation is not enough to accurately determine what would be in the register at the time of the instruction. A more advanced method would need to be developed to properly track these instructions. Additionally, a backwards propagation method was introduced for fixing jump tables in the binary. However, a large number of failed patches happened during this method of fixing jump tables. This method needs to be improved to increase the overall feasibility of using the artifacts as generic static binary rewriters.

The second area for improvement lies in the removal of ROP gadgets. As it stands, only a handful of instructions are marked for removal. The amount of instructions artifact #2 targets would need to be expanded if all mid-instruction ROP gadgets were to be removed from the binaries. Also, the current implementation of the removal of call instructions only works for call instructions that contain the “c3” byte in the first byte of the offset calculation. If both the number of targeted instructions is increased and the call instruction removal method is improved, all the Coreutil binaries should be able to have all their “c3” mid-

instruction ROP gadgets removed. Additionally, the method could be expanded upon to target ROP gadgets that do not contain the “c3” byte in them. One such instance would be the “c2” byte. This would require some modifications to the current removal method because it will introduce “c2” bytes in certain instances. If the “c2” byte is targeted as well, the overall amount of ROP gadgets in the binary will be decreased.

Improvements to both the static binary rewriting methods and the ROP gadget methods will increase the accuracy and effectiveness of the tools. Improving the static binary rewriting methods will increase the overall accuracy, which will lead to a larger number of binaries that are able to be patched. Improvements to the ROP gadget removal methods will increase the amount of ROP gadgets that can be removed and thus lower the total count of ROP gadgets that remain in the binary. If improvements can be made to either or both areas, the effectiveness of the artifacts would be greatly increased.

Closing

This study analyzed the effectiveness of using static binary rewriting for ROP gadget removal. To do this, a static binary rewriter that did not use symbolization to perform rewriting was created. Another artifact was built upon this static binary rewriter to remove mid-instruction ROP gadgets that contained the “c3” byte in them. The metrics gathered for these artifacts were successful patch rate, binary size, execution speed, and ROP gadget count. These metrics were measured and discussed to determine the feasibility of using the created artifacts to remove ROP gadgets from binaries. Based on the metrics, it seems that this method of static binary rewriting can be used to remove ROP gadgets from binaries effectively.

Limitations and recommendations for future work were also outlined in this study. Both the limitations and recommendations were focused on two main areas. These two areas were the ability to perform binary rewriting and the removal of the ROP gadgets. Regarding binary rewriting, the lack of symbolization caused issues with tracking instructions that used registers in control flow instructions. This issue will need to be resolved going forward. For the removal of ROP gadgets, more instructions need to be targeted to remove all the instructions that contain the “c3” byte in them. Additionally, improvements need to be made to the method that fixes calls that contain the “c3” byte in them. Fixing these areas would be of great benefit for removing ROP gadgets with static binary rewriting.

REFERENCES

Arntz, P. (2023, March 9). *Update Android now! Two critical vulnerabilities patched*. Malwarebytes. https://www.malwarebytes.com/blog/news/2023/03/update-android-now-two-critical-vulnerabilities-patched

Bauman, E., Lin, Z., & Hamlen, K. W. (2018). *Superset disassembly: Statically rewriting x86 binaries without heuristics* [Paper presentation]. Proceedings 2018 Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2018.23300

Bendersky, E. (2023). Pyelftools [Python]. https://github.com/eliben/pyelftools (Original work published 2013)

Bernat, A. R., & Miller, B. P. (2011). Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 9–16. https://doi.org/10.1145/2024569.2024572

Bhat, R. (2019, February 10). *Return oriented programming (ROP) attacks*. Infosec Resources. https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/

Bierbaumer, B., Kirsch, J., Kittel, T., Francillon, A., & Zarras, A. (2018). Smashing the stack protector for fun and profit. In L. J. Janczewski & M. Kutyłowski (Eds.), *ICT systems security and privacy protection* (pp. 293–306). Springer International Publishing. https://doi.org/10.1007/978-3-319-99828-2_21

Cárdenas, J. (2019). *Quantitative analysis: The guide for beginners*. University of Valencia.

Cash, P., Isaksson, O., Maier, A., & Summers, J. (2022). Sampling in design research: Eight key considerations. *Design Studies, 78,* Article 101077. https://doi.org/10.1016/j.destud.2021.101077

Chamith, B., Svensson, B. J., Dalessandro, L., & Newton, R. R. (2017). Instruction punning: Lightweight instrumentation for x86-64. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 320–332. https://doi.org/10.1145/3062341.3062344

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., & Deng, R. H. (2014). *ROPecker: A generic and practical approach for defending against ROP attack* [Paper presentation]. NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium, San Diego, February 23–26. https://doi.org/10.14722/ndss.2014.23156

Chiueh, T.-C., & Hsu, F.-H. (2001). RAD: A compile-time solution to buffer overflow attacks. *Proceedings 21st International Conference on Distributed Computing Systems*, 409–417. https://doi.org/10.1109/ICDSC.2001.918971

coreutils.git—GNU coreutils. (n.d.). Retrieved March 23, 2023, from https://git.savannah.gnu.org/cgit/coreutils.git

CWE (n.d.). *CWE-120: Buffer copy without checking size of input ('Classic buffer overflow')* (4.12). Retrieved October 13, 2023, from https://cwe.mitre.org/data/definitions/120

Cybersecurity & Infrastructure Security Agency (CISA). (2021, August 20). *Top routinely exploited vulnerabilities* (Alert code AA21-209A). https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-209a

Dinesh, S., Burow, N., Xu, D., & Payer, M. (2020). RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. *2020 IEEE Symposium on Security and Privacy*, 1497–1511. https://doi.org/10.1109/SP40000.2020.00009

Duck, G. J., Gao, X., & Roychoudhury, A. (2020). Binary rewriting without control flow recovery. *Proceedings of the 41st ACM SIGPLAN Conference on Programming*

Language Design and Implementation, 151–163. https://doi.org/10.1145/3385412.3385972

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., & Sidiroglou-Douskos, S. (2015). Control jujutsu: On the weaknesses of fine-grained control flow integrity. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 901–913. https://doi.org/10.1145/2810103.2813646

Fida El-Din, A., & Krad, H. (2007). A new trend for CISC and RISC architectures. *Asian Journal of Information Technology*.

Fiscutean, A. (2022, May 10). *Zero-click attacks explained, and why they are so dangerous*. CSO Online. https://www.csoonline.com/article/572727/zero-click-attacks-explained-and-why-they-are-so-dangerous.html

Flores-Montoya, A., & Schulte, E. (2020). Datalog disassembly. *29th Usenix Security Symposium*, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

Ganz, J., & Peisert, S. (2017). ASLR: How robust is the randomness? *2017 IEEE Cybersecurity Development (SecDev)*, 34–41. https://doi.org/10.1109/SecDev.2017.19

Gatlan, S. (2023, April 7). *Apple fixes two zero-days exploited to hack iPhones and Macs*. BleepingComputer. https://www.bleepingcomputer.com/news/apple/apple-fixes-two-zero-days-exploited-to-hack-iphones-and-macs/

Ghidra Software Reverse Engineering Framework. (2023). National Security Agency. [Java]. https://github.com/NationalSecurityAgency/ghidra (Original work published 2019)

Github. (2023). angr. [Python]. https://github.com/angr/angr (Original work published 2015)

GitHub. (2023). Capstone engine [C]. https://github.com/capstone-engine/capstone (Original work published 2013)

GitHub. (2023). Radare2. [C]. https://github.com/radareorg/radare2 (Original work published 2012)

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram, & Sudha. (2004). Design science in information systems research. *Management Information Systems Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

hex-rays. (n.d.). *IDA as a disassembler*. Retrieved February 12, 2023, from https://www.hex-rays.com/ida-pro/ida-disassembler/

Hu, Y., Zhang, Y., & Gu, D. (2019). Automatically patching vulnerabilities of binary programs via code transfer from correct versions. *IEEE Access*, *7*, 28170–28184. https://doi.org/10.1109/ACCESS.2019.2901951

Kali Linux Tools. (n.d.). *Ropper*. Retrieved May 7, 2023, from https://www.kali.org/tools/ropper/

Kang, P. (2017). Function call interception techniques. *Journal of Software: Practice and Experience*, *48*(3),385–401. https://doi.org/10.1002/spe.2501

Keystone. (n.d.). *Keystone – The ultimate assembler*. Retrieved March 23, 2023, from https://www.keystone-engine.org/

Kinder, J. (Ed.). (2010). *Static analysis of x86 executables*. Technische Universität Darmstadt.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394. https://doi.org/10.1145/360248.360252

Marcho, C. (2019, March 15). *To DEP or not to DEP …* . Microsoft Tech Community. https://techcommunity.microsoft.com/t5/ask-the-performance-team/to-dep-or-not-to-dep-8230/ba-p/373137

Marczak, B., Scott-Railton, J., Razzak, B. A., & Deibert, R. (2023). *Triple threat: NSO group's Pegasus spyware returns in 2022 with a trio of iOS 15 and iOS 16 zero-click exploit chains*. Citizen Lab, University of Toronto. https://citizenlab.ca/2023/04/nso-groups-pegasus-spyware-returns-in-2022/

Margaret, R. (2015, May 11). *What is datalog?* Techopedia. http://www.techopedia.com/definition/3915/datalog

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80–83. https://doi.org/10.1109/MSECP.2004.1281254

Newline. (2021, December 26). *A practical guide to GCC inline assembly*. https://blog.alex.balgavy.eu/a-practical-guide-to-gcc-inline-assembly/

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., & Kirda, E. (2010). G-Free: Defeating return-oriented programming through gadget-less binaries. *Proceedings of the 26th Annual Computer Security Applications Conference*, 49–58. https://doi.org/10.1145/1920261.1920269

Patterson, D. A., & Sequin, C. H. (1981). RISC I: A reduced instruction set VLSI computer. *Proceedings of the 8th Annual Symposium on Computer Architecture*, 443–457.

Planet, C. (2010, November 17). *Phrack*. http://phrack.org/issues/67/9.html

Prasad, M., & Chiueh, T. (2003). *A binary rewriting defense against stack-based buffer overflow attack* [Conference presentation]. USENIX 2003: Annual Technical

Conference, San Antonio, Texas. https://www.usenix.org/conference/2003-usenix-annual-technical-conference/binary-rewriting-defense-against-stack-based

Randell, B. (1969). A note on storage fragmentation and program segmentation. *Communications of the ACM*, *12*(7), 365–ff. https://doi.org/10.1145/363156.363158

ROP Emporium. (n.d.). *ROPEmporium*. Retrieved October 15, 2023, from https://ropemporium.com/guide.html

Rostami, M., Koushanfar, F., & Karri, R. (2014). A Primer on Hardware Security: Models, Methods, and Metrics. Proceedings of the IEEE, 102(8), 1283–1295. https://doi.org/10.1109/JPROC.2014.2335155

Ruan, Y., Kalyanasundaram, S., & Zou, X. (2016). Survey of return-oriented programming defense mechanisms: Survey of return-oriented programming defense mechanisms. *Security and Communication Networks*, *9*(10), 1247–1265. https://doi.org/10.1002/sec.1406

Saleh, M. (2020). An anatomy of Windows Executable File (EXE) and Linux Executable and Linkable Format File (ELF) formats for digital forensic analysis and anti-virus design purposes. *IJARCCE*, *7*(1), 78–84. https://doi.org/10.17148/IJARCCE.2018.71101

Sandelowski, M. (1995). Qualitative analysis: What it is and how to begin. *Research in Nursing & Health*, *18*(4), 371–375. https://doi.org/10.1002/nur.4770180411

sash. (2014, August 31). Ropper—Rop gadget finder and binary information tool. Scoding.De. https://scoding.de/ropper/

Schulte, E., Brown, M. D., & Folts, V. (2022). A broad comparative evaluation of x86-64 binary rewriters. *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, 129–144. https://doi.org/10.1145/3546096.3546112

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

Sharif, M. H. U., & Mohammed, M. A. (2022). A literature review of financial losses statistics for cyber security and future trend. *World Journal of Advanced Research and Reviews*, *15*(1), 138–156. https://doi.org/10.30574/wjarr.2022.15.1.0573

Soma, Y., Gerofi, B., & Ishikawa, Y. (2014). Revisiting virtual memory for high performance computing on manycore architectures: A hybrid segmentation kernel approach. *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, 1–8. https://doi.org/10.1145/2612262.2612264

Sporici, D. (2019, July 2). *Bypassing ASLR and DEP – Getting shells with pwntools*. Coding.Vision. https://codingvision.net/bypassing-aslr-dep-getting-shells-with-pwntools

Aleph One. (n.d.) Smashing the stack for fun and profit. *Phrack*, *7*(49), Article 14. https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Turley, J. (2014). *The basics of Intel® Architecture*. Intel. https://www.intel.com/content/www/us/en/intelligent-systems/embedded-systems-training/ia-introduction-basics-paper.html

vom Brocke, J., Hevner, A., & Maedche, A. (2020). Introduction to design science research. In *Design science research. Cases* (pp. 1–13). Springer Link. https://doi.org/10.1007/978-3-030-46781-4_1

Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., & Vigna, G. (2017). *Ramblr: Making reassembly great again* [Paper presentation].

Proceedings 2017 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2017.23225

Wang, S., Wang, P., & Wu, D. (2015). Reassembleable Disassembling. *24th Usenix Security Symposium*, 627–642. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai

Wang, S., Wang, P., & Wu, D. (2016). *UROBOROS: Instrumenting stripped binaries with static reassembling* [Conference presentation]. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan. https://doi.org/10.1109/SANER.2016.106

Wenzl, M., Merzdovnik, G., Ullrich, J., & Weippl, E. (2019). From hack to elaborate technique—A survey on binary rewriting. *ACM Computing Surveys*, *52*(3), 1–37. https://doi.org/10.1145/3316415

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer. https://doi.org/10.1007/978-3-662-43839-8

Xu, S., & Wang, Y. (2022). Defending against Return-Oriented Programming attacks based on return instruction using static analysis and binary patch techniques. *Science of Computer Programming*, *217*, Article 102768. https://doi.org/10.1016/j.scico.2022.102768

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* [Paper presentations]. 2020 International Symposium on Theoretical Aspects of Software

Engineering (TASE), Hangzhou, China. https://doi.org/10.1109/TASE49443.2020.00036

Zhang, M., & Sekar, R. (2013). Control Flow Integrity for COTS Binaries. *22nd Usenix Security Symposium*, 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., & Gu, D. (2017). *Embroidery: Patching vulnerable binary code of fragmentized android devices* [Paper presentation]. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China. https://doi.org/10.1109/ICSME.2017.15