OULUN YLIOPISTO
UNIVERSITY of OULU

A Brief Tour On Control-Flow Protection

University of Oulu
Faculty of Information Technology and Electrical Engineering
Degree Programme in Information Processing Science
Bachelor’s Thesis
Aleksi Backman
22.04.2019

Abstract

The purpose of this work is to give an overview on the topic under discussion, control-flow protection. An effort is made for the result to be more accessible by providing sufficient background in beginning and related material in the end. Most of the work was done by searching, consuming and referring to relevant research material. Additionally a control-flow integrity feature of Clang compiler is tried out and the results reported. Control-flow protection can be attacked in various ways on multiple levels and this makes it challeging to implement a trustworthy protection. For this reason it is important to understand the topic both in depth and breadth.

Keywords

Control-flow, Clang, Instrumentation, Hypervisor, Security, Buffer Overflow

Contents

1 Introduction

In this work the goal is to discover and introduce different kinds for methods for kernel integrity protection. Particularly I want to provide a very basic understanding on kernel rootkits: how they operate and how to protect against them. There are two main techniques that I will focus on: hypervisors and code instrumentation. These techniques are chosen as I initially evaluate them to be practical and hard to subvert. An ideal but hardly achieveable outcome would be to gain knowledge for building a truly secure computer system.

Another goal of this work is raising awareness and spiking interest. If a single reader wants to read more or learns something new from computer security, then I already consider the result as a success.

The target audience is inexperienced system programmers, such as myself, with some knowledge of operating system internals. If presented content feels inaccessible, please take a look at the further reading section (5.4) at the end of this work. This whole work can be thought of (and read) as a springboard to more information.

Recommended album to listen while reading is Ace by Huey.

1.1 Research problems and methods

What kind of practical threats and protection methods exist for kernel control-flow integrity? I will try to answer this question by delving into previous research on this topic as well as studying existing project(s) implementing kernel control flow integrity protection. Main focus is in leveraging previous research and minor, enriching work will be done by putting findings into practice.

I want to find out what protection techniques are still considered to be secure. My experience has been such that there are many protection mechanisms, but as time passes, some weakness gets found in them. In essence my main concern is how to make sure that a system actually performs the computations which the user expects it to. In general, dynamic control flow transfer appears to be more challenging to protect. An example is protecting hooks in kernel code compared to dynamic hooks. (Wang, Jiang, Cui, & Ning, 2009, p.2)

The research method in use is literature review. In literature review the author can explore an already well established or an emerging issue. In case of a well established field the end result would highlight the main contributions of the literature. When dealing with an emerging issue, it can be exposed to existing theory. On a high level the work consists of finding relevant research, comparing and combining results and writing the actual review. The author should be well acquainted in the topic at hand in order to be able to extensively inform collegues and provide a beneficial direction for future work. (Webster & Watson, 2002)

1.2 Restrictions for research

I will focus on Linux and its kernel whenever possible. Linux is appealing because of being widely used as well as being an open source project. Open source code makes the implementation of Linux more effortless to learn and experiment with when compared to closed source alternatives.

2 Background

Topics for providing context to this paper are covered in this section. First some central control-flow related threats are introduced on a high level. Then identified main protection methods are touched upon.

2.1 Threats

Threats for control-flow integrity are discussed next. Buffer overflows are a major software vulnerability that acts as an important building block for many attacks. There are many flavors of buffer overflows but a more detailed description of them is outside the scope of this work. Return-oriented programming is a method that makes it more difficult to prevent exploiting a buffer overflow.

2.1.1 Buffer overflows

In buffer overflow, more data is written to a variable than its defined size is. If not otherwise restricted, this extra data will fill memory next to the variable that was targeted with the write. Previous content of memory will be overwritten. This makes it possible to set any values into memory. Overwriting regular values, pointer values and control data such as function return addresses becomes possible. Pointer rewriting makes it point to a wanted memory address. Same is true for rewriting function return address. Afterwards the program execution will jump to a wanted address when the function returns. (One, 1996, pp.5-6)

2.1.2 Return-oriented programming

A way of exploiting buffer overflow is to write executable (CPU architecture specific) code to stack memory and overwrite return address of a function so that it points to the beginning of injected code. It is possible to protect a system from this exploit (to a degree) by making stack non-executable. With return-oriented programming (ROP) it is possible to achieve arbitrary code execution even when non-executable stack is in place. In ROP, already existing code is used instead of injected code. Much useful code (such as libraries, including libc) is already loaded to memory. This code can be used for attacker's purposes by chaining wanted parts of code (gadgets) together. Each gadget needs to end in a return statement in order to make it possible to stay in control of the program execution. By using the overflow, an attacker can write stack frames that eventually perform any wanted computation. (Hund, Holz, & Freiling, 2009, pp.2-3; Arpaci-Dusseau & Arpaci-Dusseau, 2018, 275)

2.1.3 Rootkits

Rootkits aim to make long-term misuse of target possible. A way of achiewing longevity is to provide means for reentry. Rootkits carry out reconnaissance by intercepting private data such as keypresses and network traffic. They can try to make defence mechanisms ineffective by neutralizing or making themselves invisible to detection methods. The previous goals of rootkit are made possible or easier by being successful in escalating privileges of the rootkit. Further, user-space object hiding and privilege escalation make it possible to have part of the rootkit implementation in user-space. (Petroni Jr & Hicks, 2007, p.2,p.4) Some motivating aspects for user-space implementation are simplicity and reliability. After all, the goal is to stay in operation for a long time.

Kernel object hooking (KOH) and dynamic kernel object modification (DKOM) are two categories for kernel rootkits. KOH rootkits try to gain control of execution by modifying code hooks or, usually, data hooks. In practice, most data hooks are function pointers. DKOM rootkits perform modifications to data that does not directly affect control flow. (Wang et al., 2009, p.2) An example is removing a process from a structure used in process accounting but keeping it in a structure that is used for process scheduling. Effectively this hides the process from programs, such as *ps*, that use the accounting mechanism, but keeps the hidden process receiving time on CPU. (Petroni Jr, Fraser, Walters, & Arbaugh, 2006) Of these two types, KOH rootkits are more common. (Wang et al., 2009, p.2)

2.2 Protection

There is a need for protection of control flow integrity. It remains an issue even though there has been progress in development of protection methods. (Abadi, Budiu, Erlingsson, & Ligatti, 2005, p.1; Davi, Sadeghi, & Winandy, 2009, p.1) There is no single way for countering the mentioned threats. The sections below cover approaches for integrity protection. Of these, monitoring and instrumentation solutions are discussed in more detail later in the work.

2.2.1 Hardening

Gentoo Linux distribution offers a hardened profile. It contains a collection of changes for system software. These changes include building a toolchain with security enhancing options enabled, PaX extensions, SELinux extensions and offline integrity protection. PaX includes address space layout randomization which is discussed below. (Gentoo Foundation, Inc., n.d.) Gentoo is only given as an example due to the collection of multiple hardening changes in the mentioned profile. Hardening changes can be applied on other distrubutions as well.

Address space layout randomization is a technique that is used for protecting against ROP attacks. The idea behind this defense is that attacker no longer knows the virtual address where a useful gadget has been loaded. (Arpaci-Dusseau & Arpaci-Dusseau, 2018, p.275; Göktas, Athanasopoulos, Bos, & Portokalidis, 2014, p.12) Naturally this defence provides no protection if the attacker is able to find out the address regardless of randomization. Exploits for leaking out crucial memory content do exist. (Göktas et al., 2014, p.12)

Additional hardening methods are shadow-stack and return address encryption. (Carlini & Wagner, 2014, p.13)

2.2.2 Monitoring

There exist a multitude of hypervisor-based monitoring solutions. They are implemented on different hypervisors. KVM, Qemu and Xen are some of the platforms encountered during this research. The monitoring solutions inspect and restrict targeted guest operating system behavior. Hypervisors provide a way of isolating monitoring program from the target. If done properly, the monitoring system will stay in operation even in the case of guest operating system being compromised. (Wojtczuk, 2008, p.1)

2.2.3 Instrumentation

Instrumentation is a technique that is used to inspect and alter program behavior during execution. This is made possible by inserting additional code for this purpose. In this work instrumentation is considered from the control-flow protection point of view. Runtime checks and breach mitigating responses can be added by instrumenting the code to protect. (Luk et al., 2005, p.1) Control-flow integrity violation can be addressed by terminating the hostile process. (The Clang Team, n.d.)

3 Prior research

Discovered protection methods and threats are next discussed in more detail. The selected examples were chosen as they touch different central aspects either from attacking or defending side.

3.1 Threats

Threats are covered from a couple of perspectives. History hiding attack is discussed in more detail to show where weakness may be found in a control-flow protection method. ROP automation shows a practical way of carrying out a ROP attack. Rowhammer and malicious BIOS modification are threats which operate beyond the protection methods part of this work.

3.1.1 Return-oriented programming

ROP automation A toolset has been developed for making it easier to develop and deploy ROP executables. A developer no longer needs to find useful gadgets manually. The toolset consists of constructor, compiler and loader. Constructor finds useful instructions and uses them to generate gadgets. Compiler uses new special-purpose source code and the constructed gadgets to create a return-oriented program. This program is then prepared for execution by loader. Loader produces absolute addresses for the given program. Loader is intended to be used by linking it to an exploit. (Hund et al., 2009, p.4)

Software development when using ROP can easily be more complex and time-consuming than software development without it. This makes the attacks less practical, although still effective. A toolset for automating much of the time-consuming work makes the attack more practical and proves that it is worth of attention.

History hiding attack So-called history hiding attack is an improvement to ROP. This attack evades ROP detection that operates by identifying sucpiciously large amounts of subsequent short gadgets in recent history. Ensuring that returns lead to call-preceded instructions is another detection method that is defeated by this ROP variation. (Carlini & Wagner, 2014, p.2) Term call-preceding instruction, is used to refer to an instruction that immediately follows a call instruction. (Carlini & Wagner, 2014, p.3) History hiding attack is used to avoid monitoring program kBouncer, which makes both of these checks when a system call is about to take place. The checks are performed for execution history entries found in Last Branch Record (LBR). LBR is a feature found in recent Intel CPUs. It keeps track of latest indirect branches made during execution. (Carlini & Wagner, 2014, p.4)

Steps of history hiding attack are:

1. Initial ROP attack
2. Clearing LBR history
3. Evading length-based gadget detection
4. Restoring register values and executing the system call

In history hiding attack, traditional ROP is first used to setup memory and registers. Registers are set to wanted values for system call invocation. This step does not avoid detection in itself. This can be done due to kBouncer checking history only when a system call is executed. No check is made yet. (Carlini & Wagner, 2014, p.5)

Second, LBR history is flushed by using call-preceded gadgets that execute indirect branches. The gadgets should modify as few registers as possible to make last step, restoring registers and performing the system call, more simple. Gadgets ending in ret instruction were used. Afterwards, LBR history contains only known entries that are all call-preceded. This satisfies one of the checks done by kBouncer. All of the used gadgets were short, so this is not yet enough to avoid detection. (Carlini & Wagner, 2014, p.5)

Third step deals with detection method that tries to catch a ROP attack by ensuring that 8 most recent indirect branches do not belong to sequences consisting of 20 or less instructions. Sequences in short history flushing gadgets would get caught by this method. If at least one of the 8 sequences is more than 20 instructions, then the history is deemed by kBouncer to not contain a ROP attack. This detection method is avoided by executing a gadget that is call-preceded and longer than 20 instructions. This gadget should again modify as few registers as possible and end by keeping attacker in charge of control-flow. The long gadget is named as *termination gadget*. (Carlini & Wagner, 2014, pp. 4-6)

Final step is restoring register values and executing the system call. If register values were modified during previous steps, they now need to be restored to the state achieved by initial setup. A preferred method was to use gadgets that end in indirect calls. These are calls made to locations which are not known at compile-time and oftentimes the target address of call is stored in regular (i.e. RAM) memory. This allows hijacking wanted calls during initial setup step and later using them in this restoration step. State restoring gadgets need to be call-preceding and they can be short. A restriction is to use at most 7 gadgets in this step. This limitation comes from the termination gadget residing in LBR history. Its purpose is to allow usage of short gadgets. Using more than 7 gadgets will push the terminating gadget outside of 8 most recent LBR entries which are checked by kBouncer. Other possibility would be to use a long gadget during register restoration, but longer gadgets are more challenging to control. (Carlini & Wagner, 2014, pp. 6-7)

The system call was usually made by calling its wrapper function (in libc, for example). The start of a system call wrapper function is not call preceded, so it is not possible to simply use a return in guiding execution there. One method was to use a call-preceding gadget ending in an indirect jump getting its destination address from a register. A return could be made to the start of the gadget. Getting to the start of the wrapper function with the indirect jump is legal as only returns are checked for being call-preceded. Even if it were possible to have an infinitely long LBR, it could also be circumvented by a variation of this attack. (Carlini & Wagner, 2014, pp. 7)

3.1.2 Rowhammer

“A fundamental assumption in software security is that a memory location can only be modified by processes that may write to this memory location.” (Gruss, Maurice, & Mangard, 2016, p.1) This statement shows that some aspects of (control-flow) security will be in need of reconsideration. None of the previously introduced protection methods help against this attack. All of the methods share the quoted assumption.

Denser DRAM leads to the possibility of altering memory in addresses close to the targeted addresses. This is caused by the presence memory disturbance errors. In other words, memory can be altered without directly accessing it. (Kim et al., 2014, p.1) This allows modifying memory which is stored in an area guarded with memory protection. (Kim et al., 2014, p.2) Thus any system relying on memory being restricted with read-only permissions, is susceptible to attacks committed with the use of memory disturbance errors.

One mitigating solution is accessing adjacent row of memory cells with a set probability. This can be implemented by a memory controller. The extra access may be done after a row has been opened and closed. This should significantly reduce the occurrance rate of memory disturbance errors. (Kim et al., 2014, p.11)

3.1.3 Trusting BIOS

In practice, BIOS fully controls the loading of an operating system. This leaves an opening for subverting any protection methods that an OS may later deploy. BIOS can contain backdoors and there may exist a possibility for modifying its firmware in an unauthorized manner. BIOS has full access to hardware and this opens another avenue for attacks. (Rutkowska, 2015, pp. 8-9)

Malicious BIOS is another threat that is not addressed by any of the introduced protection solutions. Methods are used to apply protection on hypervisor level, and levels above it. Maliciously modifying an OS enables the attacker to operate beyond the reach of these methods which act via the capabilities provided by the OS or hypervisor.

3.2 Protection

Monitoring and instrumentation solutions are covered next. Both of these are a way for providing runtime protection.

3.2.1 Monitoring

An advantage of monitoring via a hypervisor is being isolated from the protected OS. It is more diffficult to subvert both an OS and its hypervisor. There exist solutions which can be said to either analyze, detect or prevent rootkits. (Wang et al., 2009, p.1) HyperSafe aims to provide protection and OSck is concerned with detection.

OSck An approach taken with OSck is to use a monitoring program (OSck) that is ran by host kernel in a hypervisor. Hypervisor also runs the guest kernel which is being monitored. Guest kernel and monitoring program are separated by hypervisor and they are executed concurrently. Most of the time guest kernel state reads performed by monitoring program are not synchronized with the guest kernel. (Hofmann, Dunn, Kim, Roy, & Witchel, 2011)

The monitoring program makes an effort to protect so-called static, persistent and dynamic data of kernel under monitoring. Static data is data residing in kernel text. Persistent data is data which does not change after kernel initialization. Example of persistent data is system call table in memory as well as hardware registers which affect SYSCALL instruction. (Hofmann et al., 2011) I take it that at least registers IA32_EFER, IA32_LSTAR and IA32_CSTAR (described in Intel manual volume 4) are meant here. (Intel Corporation, n.d.) Both static and persistent data is guarded by disallowing modification after kernel initialization. (Hofmann et al., 2011)

Dynamic data (mainly function pointers) is guarded by ensuring type-safety. Function pointers in kernel heap are checked to point to functions which are considered being safe. First requirement for safe functions is the need to be part of kernel text and the text is static data which is protected against modifications in the previously mentioned manner. Second requirement is to have identical return type as well as the amount and type of arguments. (Hofmann et al., 2011)

HyperSafe As in any software, there can be flaws in a hypervisor. A proposition for hypervisor protection is HyperSafe system. It implements a *non-bypassable memory lockdown* technique. It is used for protecting memory where hypervisor code and read-only data are stored. This memory protection is further used for covering control data as well. Control data protection is done by restricting indirect control transfers to targets on a control-flow graph. These targets are stored in memory as tables and protected with the memory lockdown. (Wang & Jiang, 2010, pp. 1-2, p.5)

3.2.2 Instrumentation

Two concrete ways of injecting additional code are binary rewriting and modification during compilation. In binary rewriting, a given executable binary is changed by adding new code to sufficient locations. Binary rewriting makes it possible to perform instrumentation without having the access to source code. (Abadi et al., 2005, p.2) Vulcan by Microsoft is one of the tools in use for binary rewriting. (Tice et al., 2014, p.3; Abadi et al., 2005, p.6) Option for enforcing control-flow integrity during compilation has been implemented to Clang compiler that is part of LLVM project. (The Clang Team, n.d.; Tice et al., 2014, p.2)

Pin An interesting variation of binary rewriting is introduced by Pin. It performs just-in-time-instrumentation by rewriting necessary parts of program in order inspect it and stay in control of execution. Thus, Pin provides flexibility by being able to attach to a process and detach from it. The ability is similar to one of a debugger. (Luk et al., 2005, p.1)

Clang control-flow integrity Clang compiler makes it possible to perform instrumentation for control-flow integrity enforcement during compilation. More accurately the instrumentation happens during link-time optimization (LTO). Due to the design of Clang, this makes instrumentation independent of high-level language in use. (Tice et al., 2014, p.6) During LLVM compilation phases (including LTO) intermediate representation (IR) is the format in use. IR is a generalized assembly language for LLVM. When compilation is in progress, first a high-level language specific front-end (such as Clang for C language) transforms the code into IR. LTO is eventually performed on the IR. (LLVM Project, n.d.-b; LLVM Project, n.d.-a)

One control-flow integrity protection scheme addresses indirect function-calls (The Clang Team, n.d.) and it is the only scheme to be covered in this work. During LTO each indirect call is transformed so that it will reach its intended destination only by traveling through a generated jump table. These jump tables are used to force the control-flow to reach only known destinations. The jumps point to starting addresses of functions. Jump tables are located in read-only memory and this makes them more secure when compared to uninstrumented calls. Instrumentation causes each of the pointers used in indirect calls to be masked so that they to point to a (protected) jump table. (Tice et al., 2014, p.6)

4 Experiment: Instrumentation in Clang

This experiment was carried out in order to verify that the Clang control-flow integrity (CFI) enforcement works as expected. I wanted to solidify my interpretation of the workings of this feature. Another goal was to provide simple examples of basic usage and the consequent results as to make the feature easier to grasp on a concrete level. I chose this subject because it seemed very practical and straightforward. Practicality was one of the goals of the designers so I consider the tool as a success in this regard as well.

In the experiment, I compile a C program containing an indirect call with control-flow integrity protection and debug information enabled. I proceed to inspect the result with the help of gdb debugger. The target processor architecture is x86_64.

CFI protection is inspected with the help of two similar examples. At first the result of compiling a C program (containing an indirect call) with CFI protection is compared to the resulf of compiling the same program without CFI protection. Afterwards a slightly more vulnerable example of the C program is given. This is done to emphasize the need for protected jump tables which isn't as apparent in the first example.

Each compilation is performed with a couple of common flags. The common flags:

- add debug information (-g)
- select lld as the linker (-fuse-ld=lld)
- perform link-time optimization (-flto)

The commands were kept similar to try and provide a solid base for comparison. Debug information was added to make debugger usage easier. Link-time optimization was used because CFI protection requires it and lld was used for linking because it supports link-time optimization and was easily obtainable from Arch Linux package repositories. Source code of the program to be instrumented follows.

Listing 1. Program to instrument

```c
int get_two(void) {
        return 2;
}

int main(void) {
        int result = -1;
        int (*run)(void) = 0;

        run = get_two;

        if (0 != run) {
                result = run();
        }

        return result;
}
```

From CFI point of view, the interesting part of above source listing 1 is the (indirect) call made by using a function pointer on line 12. Following snippets show the surroundings of this call as well as relevant context. Here CFI protection was not used when compiling. The command used was: *clang -g -fuse-ld=lld -flto hello.c*.

Listing 2. Disassembly of function targeted by a call when no instrumentation done

```
0x201170 <+0>:     push   rbp
0x201171 <+1>:     mov    rbp,rsp
0x201174 <+4>:     mov    eax,0x2
0x201179 <+9>:     pop    rbp
0x20117a <+10>:    ret
```

Listing 3. Partial disassembly of uninstrumented main function around the indirect call

```
0x201188 <+8>:     xor    eax,eax
...
0x2011a0 <+32>:    movabs rcx,0x201170
0x2011aa <+42>:    mov    QWORD PTR [rbp-0x10],rcx
0x2011ae <+46>:    cmp    rax,QWORD PTR [rbp-0x10]
0x2011b2 <+50>:    je     0x2011ba <main+58>
0x2011b4 <+52>:    call   QWORD PTR [rbp-0x10]
0x2011b7 <+55>:    mov    DWORD PTR [rbp-0x4],eax
0x2011ba <+58>:    mov    eax,DWORD PTR [rbp-0x4]
```

Listing 4. Wrapped snippet of gdb command *maintenance info sections*

```
0x00201000->0x002011c3 at 0x00001000:
            .text ALLOC LOAD READONLY CODE HAS_CONTENTS
```

In listing 3 the address of targeted function is part of the program text which is already located in read-only memory as can be seen from listing 4. The address of targeted function (get_two disassembled in listing 2) is eventually written to stack and it is accessed from there when called. With no active protection in use, this stack value could have been changed to a malicious address before the call is executed.

Following disassembly snippets show the surroundings of the indirect call when CFI protection was used when compiling. The command used for compiling was: *clang -g -fuse-ld=lld -flto -fsanitize=cfi -fvisibility=hidden hello.c*.

Listing 5. First instruction of function targeted by a call when instrumentation done

```
0x201170 <+0>:     push   rbp
```

Listing 6. Disassembly of jump table

```
0x2011e0 <+0>:    jmp    0x201170 <get_two>
0x2011e5 <+5>:    int3
0x2011e6 <+6>:    int3
0x2011e7 <+7>:    int3
```

Listing 7. Partial disassembly of instrumented main function around the indirect call

```
0x201188 <+8>:    xor    eax,eax
...
0x2011a0 <+32>:   movabs rcx,0x2011e0
0x2011aa <+42>:   mov    QWORD PTR [rbp-0x10],rcx
0x2011ae <+46>:   cmp    rax,QWORD PTR [rbp-0x10]
0x2011b2 <+50>:   je     0x2011ce <main+78>
0x2011b4 <+52>:   mov    rax,QWORD PTR [rbp-0x10]
0x2011b8 <+56>:   movabs rcx,0x2011e0
0x2011c2 <+66>:   cmp    rax,rcx
0x2011c5 <+69>:   je     0x2011c9 <main+73>
0x2011c7 <+71>:   ud2
0x2011c9 <+73>:   call   rax
0x2011cb <+75>:   mov    DWORD PTR [rbp-0x4],eax
0x2011ce <+78>:   mov    eax,DWORD PTR [rbp-0x4]
```

Listing 8. Wrapped snippet of gdb command *maintenance info sections*

```
0x00201000->0x002011e8 at 0x00001000:
        .text ALLOC LOAD READONLY CODE HAS_CONTENTS
```

In the instrumented result of compilation, the address of indirect call is again part program text. In listing 7, the address to be used in call (0x2011e0) does not point directly to the targeted function (in listing 5) at 0x201170. Instead, it points to a jump table (in listing 6) which contains a jump to the targeted function. The table is padded with int3 instructions which have the default behavior of terminating the process. All of the code above resides in read-only memory as can be seen from listing 8.

The address to call is eventually loaded from stack to to rax register. The instrumentation causes rax to be compared to rcx which contains the address to call. If stack were modified and malicious address was loaded to rax, execution of ud2 would happen and the process is killed in a controlled manner. The jump table contains addresses of known legitimate functions. Forcing the execution to travel through the read-only jump table is used in guaranteeing that known functions are called.

An example highlighting pointer corruption is up next. A function in code of a product could well receive function pointers or indices to function pointers as parameters.

Listing 9. Instrumented program with potential for pointer corruption

```c
#include <stdlib.h>

int get_two(void) {
        return 2;
}

int main(int argc, char* argv[]) {
        int corrupt = 0;
        int result = -1;
        int (*run)(void) = 0;

        run = get_two;

        if (argc > 1) {
                corrupt = atoi(argv[1]);
        }

        run = run + corrupt;
        if (0 != run) {
                result = run();
        }

        return result;
}
```

Listing 10. Partial disassembly of instrumented main function with potential for pointer corruption

```
0x21bc <+44>:    lea     rax,[rip+0x5d]        # 0x2220 <get_two>
0x21c3 <+51>:    mov     QWORD PTR [rbp-0x8],rax
...
0x21d5 <+69>:    call    0x2270 <atoi@plt>
0x21da <+74>:    mov     DWORD PTR [rbp-0x10],eax
0x21dd <+77>:    xor     eax,eax
0x21df <+79>:    mov     rcx,QWORD PTR [rbp-0x8]
0x21e3 <+83>:    movsxd  rdx,DWORD PTR [rbp-0x10]
0x21e7 <+87>:    add     rcx,rdx
0x21ea <+90>:    mov     QWORD PTR [rbp-0x8],rcx
0x21ee <+94>:    cmp     rax,QWORD PTR [rbp-0x8]
0x21f2 <+98>:    je      0x220b <main+123>
0x21f4 <+100>:   mov     rcx,QWORD PTR [rbp-0x8]
0x21f8 <+104>:   lea     rax,[rip+0x21]        # 0x2220 <get_two>
0x21ff <+111>:   cmp     rcx,rax
0x2202 <+114>:   je      0x2206 <main+118>
0x2204 <+116>:   ud2
0x2206 <+118>:   call    rcx
```

Listing 11. First instruction of jump table

```
0x2220 <+0>:      jmp     0x2180 <get_two>
```

Listing 12. First instruction of function targeted by a call

```
0x2180 <+0>:      push    rbp
```

Listing 13. Wrapped snippet of gdb command *maintenance info sections*

```
0x00002000->0x00002228 at 0x00002000:
        .text ALLOC LOAD READONLY CODE HAS_CONTENTS
```

Only disassembly from instrumented executable is shown in above listings. Again, the call to targeted get_two (in listing 12) is made through jump table (in listing 11) and all code, including the jump table, is in read-only memory (as seen in listing 13).

In main function of the program with possibility for pointer corruption (listing 9), the modification to pointer is received as an argument. This provided a relatively clear way for testing and portraying function pointer corruption. With no arguments the program finished successfully. When ran with arguments, the pointer to get_two is altered and execution stops prematurely. A method closer to an actual scenario would have been to modify values on stack with gdb.

When running the instrumented executable for testing and the function pointer was corrupted, execution ended to illegal instruction in a controlled fashion. This was due to execution of ud2 (found in listing 10). Uninstrumented executable ended in a segmentation fault and this shows the possibility for exploitation. Some illegal code can be executed before the segmentation fault and if crafted appropriately, no segmentation fault will occur.

I did not perceive the mentioned masking of indirect calls. I expect this to be due to program containing only a single indirect call. During experimentation I already saw compilation results with subtle differences. I expect that Clang is making adjustments and tries to simplify as much as possible. I would assume the masking to happen at the point where (in the examples) rcx is compared to rax just before conditionally jumping over ud2.

5 Discussion

At the moment I see no reason why monitoring and instrumentation could not be used simultaneously to gain benefits from both approaches. A draft of a system like this is shown in figure 1, where the solid singly pointed arrows represent loading and doubly pointed arrows represent communication. A possible challenge could be performance, however, many solutions already offer their protection with a low overhead. I would prefer instrumenting during compilation instead of binary rewriting. Using a compiler is already part of software development so it would not be a huge departure from an already established way of working.

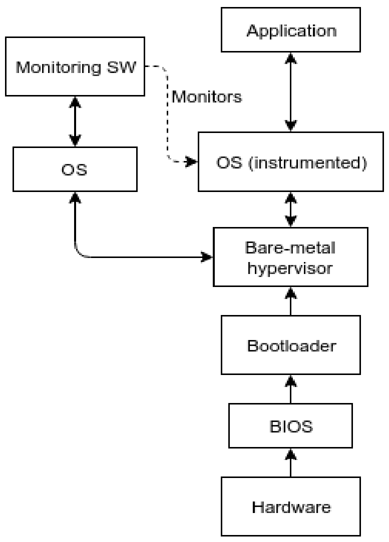

Figure 1. Highly simplified relations of a system where monitoring and instrumentation are both in use.

Clang CFI feature trusts in protection provided by read-only memory. Read-only permission can be bypassed by rowhammer or BIOS as it has unrestricted access to hardware. In itself the CFI feature is strong, but in the context of the entire system it comes off as breakable. In my opinion this shows the need for combining different protection solutions targeting different threats. These threats may or may not be on different levels on the overall components (software/firmware/hardware) of the system. I feel that it would be unreasonable to expect the CFI feature, for instance, to cope with attacks utilizing hardware related vulnerabilities. Were it possible, doing so would further complicate the implementation and consequently make it more brittle. The issues should be solved on an appropriate level.

5.1 Research process

Before starting to write at all, I dreaded the idea of having to use Microsoft Office tools after having gotten used to editing text with Vim. Another concern beside more clunky key bindings was the inability to use a version control system for backup, distribution, history branching and viewing. Git, the version control system of choice, does not support showing differences between binary files. Handling binary files with Git is generally discouraged. At least for me it makes sense to store text in the form of text files.

I found out that it possible to render formatted pdf files from text files by using LaTeX and its syntax. This allowed me to edit pure text files with any regular text editor as well as keeping the work under version control.

At the beginning of research it was not clear what the main focus would be. I had read about an interesting project, Linux Kernel Runtime Guard (LKRG), and wanted to know what it does when trying to keep the kernel secure during execution. I started to find literature by searching work that is related to LKRG. My plan was to perform an experiment by using LKRG. Along the way I found other projects that perform kernel monitoring.

Other projects had presented ideas of running the monitoring software inside a hypervisor that is used to run the guest kernel as well. Isolating monitoring software from kernel under inspection did seem like a more secure approach to me. One implementation ran the monitoring software on a separate PCI card and inspected target kernel by reading RAM content and making observations from the acquired data. Now LKRG seemed less appealing because of it being a loadable kernel module and thus eventually part of the running kernel.

The papers had mentioned rootkits and described their behavior. I started to see that I specifically wanted to find out more about rootkits in general and how to protect a computer system from them.

Eventually I ran into a blog post about Android runtime integrity protection. The tool used for protection was Clang compiler. Google had implemented the integrity protection feature and provided links to relevant LLVM project documentation. In the documentation there were links to research papers on this topic. One of them was specifically about the control-flow integrity feature in question. I started to get particularly interested about dynamic control-flow.

As I kept reading the research papers, I started to experience that many of them focus on a single method. Discussion of attacking or protecting control flow circled more or less around a single point of view. I was interested in finding out what techniques could be *combined* in order to produce a reasonably secure system.

I did remember reading about an interesting attack, rowhammer, and suspected that with this attack it would be possible to thwart all the previous protection mechanisms. I also did remember reading about BIOS vulnerabilities in the past. A search from Google Scholar gave some interesting hits. I suspected the previous assumption to be true in this case also. I may have initially read about rowhammer from the research paper ”Intel x86 considered harmful” a while back. The attack is mentioned in the paper. I no longer have a clear memory of this.

I had wanted to conduct an experiment with the Clang CFI protection when I learned of it’s existence. I came into the (wrong) conclusion that it was necessary for me to build Clang if I wanted to use the CFI feature. Eventually I did succeed in building the tool. However, CFI required link-time optimization and gold plugin was mentioned to be a viable possibility. I thought that configuring the build would consume too much time to be worth it considering the size of this work. Later, when reading the documentation again, I learned that lld does support link-time optimization. It was possible to simply use lld as the linker when compiling with CFI protection. I would consider it as a good addition to CFI documentation page to give one full Clang compilation command with lld in use. The lld linker is part of LLVM project after all. Also the Clang versions supporting this feature would be helpful or I just have not found them.

5.2 Future work

To further explore the topic, the presented system employing monitoring and instrumentation (in figure 1) could be built and evaluated. It would be enlightening to construct and describe in detail attacks against the system by using BIOS. Work could then proceed to show ways of protecting from this threat.

5.3 Conclusion

When constructing a secure system, it is necessary to consider the whole. This means implementing protection on multiple levels. Each level, all the way down to hardware, acts as a platform for the higher level to build upon.

Lastly, I want to present two more important practices for improving security through reliability. Careful review and testing can be used to reduce unexpected behavior of a program. Oversights made by programmers are at least close to the root cause of many vulnerabilities which enable some of the covered exploits. Both testing and review are made easier by taking them into consideration while designing and implementing a system. A clear solution is easier to prove as correct or incorrect.

I thank you, the reader, for your interest in my work.

5.4 Further reading

The following list contains material which has been or I predict to be valuable on the skirts of this work.

- Clean Code: A Handbook of Agile Software Craftsmanship
- Operating Systems: Three Easy Pieces (Available for free also)
- HACKING: THE ART OF EXPLOITATION, 2ND EDITION (Contains CD for learning)
- Linux kernel development, Third Edition
- Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux
- https://wiki.osdev.org/Expanded_Main_Page (OSDev Wiki)
- https://os.phil-opp.com/ (“Writing an OS in Rust”)
- https://www.coreboot.org/ (Coreboot, “replacement for your BIOS/UEFI”, open source)
- http://ts.data61.csiro.au/projects/seL4/ (seL4 microkernel)
- https://clang.llvm.org/docs/SafeStack.html (Clang SafeStack)
- https://firmware.intel.com/learn/uefi/white-papers-articles (Intel firmware resources)
- https://invisiblethingslab.com/resources/ (“virtualization, kernel and system-level security resources”)
- https://bootlin.com/docs/ (“kernel, real-time, Android, embedded Linux system and device driver development” by Bootlin)
- http://nuclear.mutantstargoat.com/articles/make/ (practical makefiles)
- https://www.kernel.org/doc/html/latest/ (Linux kernel documentation online)
- http://beej.us/guide/bgipc/ (Unix IPC learning material by Beej)
- http://asm.sourceforge.net/ (assembly language programming on Linux)
- https://pacman128.github.io/pcasm/ (“PC Assembly Language” by Paul A. Carter)
- http://www.yolinux.com/TUTORIALS/GDB-Commands.html (A source of tutorials)
- https://wiki.xenproject.org/wiki/Main_Page (Xen hypervisor)

6 Acknowledgements

I want to thank Kimmo Korhonen for providing a correctly stylized LaTeX template. This pushed me forward in writing the work.

References

Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005). Control-flow integrity.

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating systems: Three easy pieces. Arpaci-Dusseau Books LLC.

Carlini, N., & Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In 23rd usenix security symposium (usenix security 14) (pp. 385–399).

Davi, L., Sadeghi, A.-R., & Winandy, M. (2009). Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In Proceedings of the 2009 acm workshop on scalable trusted computing (pp. 49–54).

Gentoo Foundation, Inc. (n.d.). Hardened gentoo - gentoo wiki. (Retrieved 2019-03-24, from `https://wiki.gentoo.org/wiki/Hardened_Gentoo`)

Göktas, E., Athanasopoulos, E., Bos, H., & Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In 2014 ieee symposium on security and privacy (pp. 575–589).

Gruss, D., Maurice, C., & Mangard, S. (2016). Rowhammer. js: A remote software-induced fault attack in javascript. In International conference on detection of intrusions and malware, and vulnerability assessment (pp. 300–321).

Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., & Witchel, E. (2011). Ensuring operating system kernel integrity with osck. In Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems (pp. 279–290). New York, NY, USA: ACM. doi: 10.1145/1950365.1950398

Hund, R., Holz, T., & Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In Usenix security symposium (pp. 383–398).

Intel Corporation. (n.d.). Intel® 64 and IA-32 Architectures Software Developer Manuals. (Retrieved 2019-03-24, from `https://software.intel.com/en-us/articles/intel-sdm`)

Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., … Mutlu, O. (2014). Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In Acm sigarch computer architecture news (Vol. 42, pp. 361–372).

LLVM Project. (n.d.-a). Llvm language reference manual – llvm 9 documentation. (Retrieved 2019-03-23, from `http://llvm.org/docs/LangRef.html`)

LLVM Project. (n.d.-b). Llvm link time optimization: Design and implementation – llvm 9 documentation. (Retrieved 2019-03-23, from `https://www.llvm.org/docs/LinkTimeOptimization.html`)

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., … Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In Acm sigplan notices (Vol. 40, pp. 190–200).

One, A. (1996). Smashing the stack for fun and profit. Phrack magazine, 7(49), 14–16.

Petroni Jr, N. L., Fraser, T., Walters, A., & Arbaugh, W. A. (2006). An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In Usenix security symposium.

Petroni Jr, N. L., & Hicks, M. (2007). Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th acm conference on computer and communications security (pp. 103–115).

Rutkowska, J. (2015). Intel x86 considered harmful. the Invisible Things Lab.

The Clang Team. (n.d.). Control flow integrity – Clang 9 documentation. (Retrieved 2019-03-11, from `https://clang.llvm.org/docs/ControlFlowIntegrity.html`)

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., & Pike, G. (2014). Enforcing forward-edge control-flow integrity in gcc & llvm. In 23rd usenix security symposium (usenix security 14) (pp. 941–955).

Wang, Z., & Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In 2010 ieee symposium on security and privacy (pp. 380–395).

Wang, Z., Jiang, X., Cui, W., & Ning, P. (2009). Countering kernel rootkits with lightweight hook protection. In Proceedings of the 16th acm conference on computer and communications security (pp. 545–554).

Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. MIS quarterly, xiii–xxiii.

Wojtczuk, R. (2008). Xen 0wning trilogy (part 1): Subverting the Xen hypervisor. (Presented at Black Hat USA, `https://invisiblethingslab.com/resources/bh08/part1.pdf`)