On the Effectiveness of Hardware Enforced Control Flow Integrity

Austin J. Gadient
United States Air Force Academy
c18austin.gadient@usafa.edu

Abstract

Defenses such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and stack canaries have been circumvented by recent exploits. As a result, security researchers have turned towards Control Flow Integrity (CFI) to defend systems. Previous attempts to achieve CFI have tried to remain efficient and practical, but were exploitable. The NSA proposed a CFI system which integrates new hardware and program instrumentation. The purpose of this research is to assess and improve this proposal. In this paper, the system is exploited through the development of simple, vulnerable programs. It is shown to be effective in mitigating Jump Oriented Programming (JOP) attacks through an algorithm introduced as part of this work. Finally, different approaches are proposed to improve upon this system while their merits and issues are assessed.

1. Introduction

Modern program exploitation has become incredibly complex and difficult [20]. Gone are the days of simply injecting and returning to shellcode at runtime [1]. This is due to recent mitigations such as DEP, stack canaries, and ASLR [22, 32, 37, 38]. Almost every modern exploit utilizes some sort of memory leak to bypass ASLR and stack guards, allowing a code-reuse attack to be performed in order to obtain arbitrary code execution [23]. These exploits have devastating results, leading to the loss of millions of dollars and sensitive information [6]. To combat sophisticated attackers and protect critical systems for good, CFI seems to be the only option.

Because a solution has not yet been achieved, new systems to attain CFI are being developed [36]. CFI was made as a response to all control flow hijacking attacks, but recent proposals have specifically targeted return oriented programming (ROP) [35]. In response, ROP attacks have become evermore sophisticated [4, 7, 9]. The NSA has proposed a defense which adds three new Landing Point Instructions (LPI) to the x86 instruction set. These are the Call Landing Point (CLP), Jump Landing Point (JLP), and Return Landing Point (RLP) [25]. If the instruction following a direct or indirect `call`, indirect `jmp`, or `ret` does not match the previous instruction's respective landing point, then the program faults [25]. Additionally, the system includes a hardware protected shadow stack to save valid return addresses and totally eliminate ROP [25]. This addition is what makes these defenses more secure than current implementations.

This research attempts to do three things. First, find situations under which this form of CFI is exploitable. Using these situations, develop a list of requirements which are needed to exploit this system. Then, determine the feasibility of JOP attacks using instrumented binaries to find valid dispatcher gadgets [5]. Finally, suggestions to improve the system are given and evaluated. [1]

To find situations in which the system is exploitable, various simple C programs with obvious vulnerabilities were written and assessed. Based on similarities between these programs and the methods used to perform exploitation, the requirements for an exploit were established. All tests were performed on 64-bit Kali Linux 2.0 run on an Intel(R) Core(TM) i5-4300M CPU @ 2.60 GHz which uses x86-64 assembly. An instrumented `libc.so.6` library was supplied by the NSA, as well as two small, vulnerable programs. The provided programs did not include those written in C++, so the possibility of attacks on C++ programs was assessed by determining if current exploitation methods would trip the defenses [16, 34].

The exploits discovered resemble return-to-libc exploits in which arbitrary commands are issued by the attacker, but a function pointer is overwritten instead of a return address [26]. CFI is meant to work independent of all other defense mechanisms besides DEP [2]. The NSA's system is no different, so ASLR and stack canaries are not considered while determining vulnera-

[1] This research was completed as part of the INSuRE program established by the NSF.

URI: http://hdl.handle.net/10125/50585
ISBN: 978-0-9981331-1-9

bilities [25]. It is important to realize that return-to-libc attacks in which attackers invoke complete system calls such as `system("/bin/sh")`, while powerful, are not Turing complete [35]. Instead, an attacker can run arbitrary commands on a target machine which can achieve the same goals as arbitrary computation. Note that the exploits discovered do not use library functions to form Turing complete computation or ROP chains [35].

Using the algorithm outlined in Bletsh et al., a new algorithm was developed and a Python script was written. It examines a series of plain text files which contain gadgets of length 50 or less from the assembly code of 246 LPI instrumented Linux executables and libraries [5, 24]. These files came from an instrumented version of the Void Linux filesystem [17]. The script attempted to search for a valid dispatcher gadget which acts as a program counter for, and is an essential portion of, a JOP attack [5, 10]. No valid dispatcher gadgets were found in these code streams based on the algorithm, suggesting that a JOP attack is improbable against this method of defense.

Overall, this research contributes the following:

1. Multiple exploitable programs are written and explored in a 64-bit architecture. The similarities between the exploits in these programs is used to develop a list of requirements needed for an attacker to exploit a C program with the NSA's proposed mitigations in place. The possibility of attacks on C++ programs is assessed by evaluating current exploitation methods.
2. An algorithm is developed to find valid dispatcher gadgets in the code streams of 246 instrumented Linux executables and libraries. A Python script is written using this algorithm to show that no practical dispatcher gadgets exist in this framework.
3. Suggestions to improve the NSA's defenses are given and evaluated on their practicality and likely performance overheads.

The rest of this paper is organized as follows. Section 2 gives a short background on the x86-64 architecture and the NSA's proposed mitigations. Section 3 explores exploitation of programs within the confines of the NSA's mitigations. Section 4 gives a brief explanation of JOP attacks, and goes over the algorithm and script used to find dispatcher gadgets in the supplied code streams. Section 5 will explore possible improvements to the NSA's solution. Section 6 concludes the paper.

2. Background

In this section, the basic stack conventions of x86-64 assembly, as well as the `jmp`, `call`, and `ret` instructions, are explained. Afterwards, the NSA's proposed mitigations are explored.

2.1 x86-64 Basics

In x86-64 assembly the first 6 pointer and integer arguments to a function are passed by registers [8]. `RIP` is the instruction pointer, and `RDI` holds the first integer or pointer argument to a function. For example, when the function `system("/bin/sh")` is called, `RIP` will contain the address of the first instruction of `system()` and `RDI` will contain the address of the string `"/bin/sh"`. These two registers are incredibly important to an attacker, as they allow him or her to control what function is being executed and what the first argument to that function is [26]. Of course, more registers need to be controlled for functions with multiple arguments. Any arguments beyond the first six are passed using the stack which is a place in memory that stores local variables and arguments to functions. A sample stack frame is shown in Figure 1. Note that the address values in Figure 1 are arbitrary and are only used to show direction of stack growth and the number of bytes at each address.

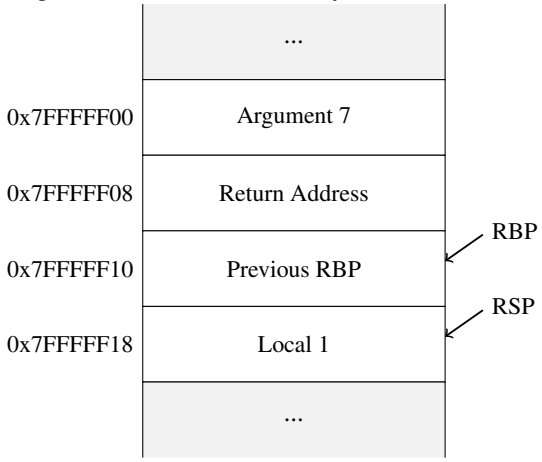

Figure 1: Sample x64 Stack

Figure 1 also shows two other important registers, the base pointer, `RBP`, and the stack pointer, `RSP`. In x86-64 the stack grows downwards and `RSP` will always point to the bottom of the stack. The base pointer points to the address of the base pointer for the previous stack frame and is used to reference local variables and arguments for the current function. When the function from Figure 1 returns, the base pointer will be set to the previous value of `RBP`, so that local variables and arguments for that function are referenced properly. The final significant part of the stack frame is the return address. When the current function finishes, `RIP` is set to the value saved in the return address.

In x86-64, there are two instructions used to alter code flow. These are the `call` and `jmp` instructions. A `jmp` instruction sets `RIP` to a specific offset from its current position. This instruction alters code flow without changing the stack or any other registers. There are various types of `jmp` instructions which change `RIP` based on certain conditions. Two notable types are indirect jumps and direct jumps. Indirect jumps are made based on the value in a memory location or register. This allows a program to alter its code flow based on the state of variables and user input. Direct jumps have a hardcoded offset that they modify `RIP` with. This hardcoded offset is determined at compile time.

In this architecture `call` instructions are very different than `jmp` instructions. There are not various types of `call` instructions based on conditions. A `call` instruction will always adjust code flow by invoking a specific function. A `call` instruction also changes the stack. If a `call` is issued, a new stack frame is created by pushing the current `RIP` location onto the stack, and an unconditional jump is performed [8]. To grow the stack, `RSP` is decremented. When a `ret` instruction is reached, `RIP` is restored to the instruction just after the previous `call`.

2.2 Proposed Mitigations

As mentioned earlier, the NSA's mitigations include three new instructions to be added to the x86 framework. These are `clp`, `jlp`, and `rlp` [25]. They have also proposed a hardware protected shadow stack which saves return addresses. Because of this, `rlp` was created simply for research and testing. It has no practical application because of the shadow stack [25].

Each LPI acts as a check for any sort of indirect branch or call. For example, if a program reaches a `call` instruction, the destination of this `call` must be a `clp` instruction. The destination of a `jmp` instruction must be a `jlp` instruction. If these requirements are violated, then the program faults, preventing continued code execution. This method, known as instrumentation, is effective because it largely prevents unintended code sequences that an attacker can use to execute malicious functionality [2].

Listing 1 shows some sample assembly code to better demonstrate how LPIs work. Say `RIP` is at the instruction on line 3. It will perform the `mov`, then it will make an indirect jump based on `RAX`. It must land on a `jlp` instruction, or the program will fault. If the instruction on line 4 transfers execution to line 6, then the program will perform the computations and call foo2. This function begins with a `clp` instruction. If it did not, the call on line 9 would cause the program to fault. `foo2` then performs its arithmetic and returns. Because `foo2` was called by the `call` on line 11, `RIP` will be set to the `rlp` instruction on line 10. Again, `rlp` is there only for research purposes to ensure that a `ret` instruction lands where it should. It has no practical purpose given the shadow stack.

```
1 foo1:
2 clp
3 mov rax, [rsp + 48]
4 jmp rax
5 ...
6 jlp
7 xor rax, rax
8 and [rsp + 48], 0
9 call foo2
10 rlp
11 foo2:
12 clp
13 mov rax, 156
14 and rax, 20
15 ret
```

Listing 1: Sample instrumented code

The shadow stack works by saving the return address pushed on to the stack after a `call` instruction to a special place in memory that is unaccessible by the program. When a `ret` instruction is reached, the value saved on the stack is checked against the value inside the shadow stack. If the two values do not match, the program faults. This prevents an attacker from altering code flow by overwriting a return address or by performing a ROP attack. ROP attacks rely on performing a small number of instructions all ended with a `ret` instruction [7, 9, 35]. As soon as one of these extraneous `ret` instructions is reached, the shadow stack will see that it is invalid and terminate the program, thereby defeating all ROP.

The NSA's proposal mixes coarse-grained CFI with fine-grained CFI. Fine-grained CFI attempts to match actual code flow with that of the intended program perfectly [2]. This, however, comes with significant overhead and is impractical using software alone [2]. Because of this, many implementations have attempted to use coarse-grained CFI [11, 31, 41]. Coarse-grained CFI sacrifices the completeness of the control-flow graph to improve performance [14, 18]. These mitigations have unfortunately been shown to be exploitable [18]. In the NSA's proposal, the new instructions provide course-grained CFI, while the shadow stack provides fine-grained CFI.

3. Exploitation

Based on manual analysis of the text files containing gadgets for the instrumented binaries supplied by

the NSA, it is clear that a traditional Call Oriented Programing (COP) or JOP attack would be incredibly difficult [5, 34]. ROP attacks are impossible because of the shadow stack. Therefore, exploitable programs were written to understand when the NSA's system is exploitable. The first program was inspired by Yang Yu's presentation at Black Hat USA 2014 [40]. The presentation involves a situation where an attacker has gained arbitrary read write, meaning they can read and write anywhere in memory. The attacker leverages this to run arbitrary system commands by escaping the code sandboxing of Internet Explorer. This approach is different than a traditional code-reuse attack, which revealed an important requirement to exploit programs protected by the NSA's mitigations. The sample program is shown in Listing 2.

To exploit the program in Listing 2, there are three steps required. First, an address from the linked library is read from memory using the format string vulnerability in line 6. Using this memory leak, the address of `system` is obtained. Then, the function pointer for `puts` in the Global Offset Table (GOT) is overwritten with the address of `system` in line 9. These two steps are only necessary when ASLR is in use. According to the NSA's specifications, their mitigations are meant to work even if ASLR is not being used [25]. Still, these steps were included to show that getting around ASLR is exactly the same with the NSA's mitigations in place.

```
1  void foo() {
2   puts(input);
3  }
4  main(int argc, char *argv[]) {
5           void (*fptr)();
6           char input[30];
7           char *intro = "Give me your
                input:";
8           puts(intro);
9           fgets(input, 30);
10          printf(input);
11          void *got_addr = 0x600b80;
12          fgets(your_str, 30);
13          memcpy(got_addr, your_str, 6);
14          fptr = foo;
15          memcpy(input, argv[2],
16          atoi(argv[1]));
17          (*fptr)();
18 }
```

Listing 2: Arbitrary Read-Write Program

The first argument for the call to `puts` in line 16 is a buffer controlled by the attacker. If the attacker supplies `"/bin/sh"` in this buffer, they can spawn a shell. This code is exploitable and fits within the bounds of the NSA's mitigations except for one problem. It shows how code flow can be changed by overwriting a function pointer, but does not show how an attacker could control arguments to functions on their own. The call to `puts` in `foo` already takes user input as its first and only argument.

To get around this roadblock, the attacker must be able to corrupt function arguments. An example of exploitable code that allows an attacker to do this is displayed in Listing 3.

To exploit this program, an attacker must simply write the string `"/bin//sh"` to `STDIN`, followed by some junk until they reach the integer pointer defined on line 4 and the function pointer defined on line 3. The integer pointer arg must be overwritten with the address of the string `"/bin//sh"`, and the function pointer `fptr` must be overwritten by the address of `system`. This will cause line 8 to implement `system("/bin//sh")`, thus giving the attacker the opportunity to execute arbitrary commands on the target system.

```
1 void foo();
2 main(int argc, char *argv[]) {
3 void (*fptr)();
4 int *arg;
5 char input[30];
6 fptr = foo;
7 scanf("%s", input);
8 (*fptr)(arg);
9 }
10 void foo(int *arg) {
11 printf("You are in foo");
12 }
```

Listing 3: Stack Corruption Program

While the executable in Listing 3 is tiny and simplistic, similar attacks have occurred on larger programs. Google Chrome was the victim of an attack where the attacker had arbitrary read write access [20]. C++ programs such as Google Chrome, Microsoft Office, and Adobe Acrobat are also prone to memory corruption vulnerabilities. C++ executables would be protected under the NSA's CFI framework as well, so it is important that their vulnerabilities are considered. This is because they suffer from the same issues as C programs. Buffer overflows, heap overflows, and use after free vulnerabilities are all present in these programs. The introduction of objects to C++ programs opens them up to a different class of vulnerabilities where objects can be hijacked for malicious purposes. This is due to the way their memory is allocated on the heap.

C++ objects with virtual functions have virtual pointers (vptrs) associated with them. A vptr is a pointer which references an object's virtual function table (vtable). The vtable is a table of function pointers for each of the functions an object can invoke. Listing

4 shows a sample C++ object. This object represents a note. The object holds its own length, the current message of the note, and allows this message to be written with the write() virtual function.

The object in Listing 4 will be allocated as shown in Figure 2. An arbitrary write vulnerability could corrupt the vptr. An attacker can hijack the vptr to point to a function pointer of their choosing instead of the proper vtable or overwrite a pointer in a vtable to gain control of `RIP`. Overwriting a pointer in a vtable has the same effect as overwriting a function pointer. To take advantage of overwriting the vptr, an attacker must set it to the location of a pointer to a desired library function. Without ASLR, it is easy to know where the function pointer's address is and the address of the desired library function. When the `write()` function is invoked after a corruption of the vptr, an arbitrary library function would be called using the `new_text` variable as an argument. This vulnerability would not be protected against by the proposed CFI system, as it allows function pointers to be overwritten.

```
1 class Note {
2 public:
3 unsigned int length;
4 char message[100];
5 virtual void write(char *new_text);
6 };
```

Listing 4: Sample C++ object

The attack described above resembles a technique known as control-flow jujutsu [16]. This attack corrupts C++ objects saved on the heap that are used as arguments to a function whose pointer is also overwritten. This allows the attacker to execute arbitrary library functions with arbitrary arguments [16]. This technique was demonstrated on Apache, a large, popular program [16].

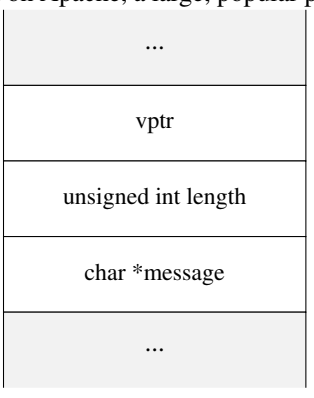

Figure 2: Note Object Allocation

Counterfeit Object-Oriented Programing (COOP) is also a technique that could be used to exploit a C++ program under the NSA's protections [34]. COOP creates gadgets by stitching together various virtual functions for different objects in large programs. Because virtual functions are valid targets for execution, there would be no violation of landing point instructions. Furthermore, this attack technique does not overwrite any return addresses. By avoiding return addresses as an attack vector, this method would overcome the shadow stack protection proposed. Another substantial strength of COOP is that it does not necessarily need to overwrite a function pointer. It can work simply be corrupting an array of objects. Because the CFI system does not prevent the corruption of memory as a whole, many types of overflows or arbitrary writes could create a counterfeit object such as this. The system only defends against overwriting return addresses and stitching together attacks with small gadgets.

The paper put out by the NSA makes the assertion that "magic gadgets" which perform all of the necessary computation for an exploit do not exist [25]. This is, however, untrue. There are singular addresses in Windows and Linux libraries that, when accessed, open up a command shell or can be used to download malicious DLLs [15, 29]. Listing 5 displays one such gadget which is available in libc.so.6.

There is only one requirement for this gadget to be used on a normal system, `[rsp + 0x50] = NULL`. If this is true, the gadget will invoke `execve("/bin/sh", NULL, NULL)`. With a shell open on the victim system and an attacker can now execute arbitrary commands.

```
1 mov  rax, QWORD PTR [rip+0x2d34bd]
2 lea  rsi, [rsp+0x50]
3 lea  rdi, [rip+0x9cb84]  # '/bin/sh'
4 mov  rdx, QWORD PTR [rax]
5 call execve
```

Listing 5: Oneshot RCE Gadget

This same codestream is in the instrumented version of libc.so.6 provided by the NSA. Fortunately, it is next to useless for an attacker. This is because of LPIs. An attacker cannot simply jump or create a call to the location of memory 0xef9f4 bytes from the base address of the library. Instead, they must first hit a CLP at the beginning of the `do_system()` function. The gadget can only be reached for its intended purpose, open up a shell for the system command that it is provided, run the command, and close the shell. Therefore, an attacker can only use this code to their advantage by calling `system` with the proper arguments. This requires control of at least two registers, `RIP` and `RDI`. The attacker must also know where `system()` resides in memory.

Based on the previously described attack vectors, a list of requirements for an attacker to exploit a program with the NSA's mitigations was obtained:

1. The attacker must overwrite a function pointer with the address of the function they would like to execute.
2. They must overwrite the arguments for the overwritten function.
3. Their memory corruption must persist until the overwritten function pointer is called.
4. If an attacker cannot do this and is attacking a C++ program, they must attempt corruption of objects to perform a control-flow jujitsu or COOP attack.

While the NSA's system is exploitable, it does successfully mitigate against ROP and JOP attacks. ROP attacks are eliminated by the shadow stack. Proof that JOP is impractical is explored in the next section.

4. Finding Dispatcher Gadgets

Jump oriented programming is an exploitation technique that does not use `ret` instructions [5]. Instead, each gadget ends in a `jmp` instruction [5]. To perform this kind of attack, a special gadget known as the dispatcher gadget is used as the program counter [5]. This gadget iterates through different functional gadgets to perform the desired computation [5]. An alternate method of JOP added a layer of gadgets that ensure each functional gadget returns control to the dispatcher gadget [10]. This method still requires a dispatcher gadget, so it is equally affected by LPIs .

The dispatcher gadget is an essential part of a JOP attack [5, 10]. This is because it stitches together each functional gadget that performs some computation interesting to the attacker. A functional gadget, by itself, will do something simple like setting `RAX` to `0xA`. An attacker would do something like this to prepare for the `mprotect` system call. To effectively use the `mprotect` function, however, the registers `RSI`, `RDI`, and `RDX` must also be set to specific values to create the proper arguments. The dispatcher gadget puts each of these functional gadgets together so that they can perform some task useful to an attacker.

An ideal dispatcher gadget is visible in Listing 6. As long as the attacker controls the memory referenced by `RAX`, they could iterate through a number of functional instructions to perform useful computation. The addresses of the functional gadgets would need to be located in the memory referenced by `RAX` for this attack to work.

```
1 add RAX, 8
2 call [RAX]
```

Listing 6: Sample Dispatcher Gadget

When JOP attacks were introduced, no instrumentation was used on the target binaries [5]. This means unintended code sequences were utilized in the attack. In the NSA's model, however, these unintended code sequences are far less prevalent. Focus was placed on trying to find a good dispatcher gadget in the text files containing gadgets from various libraries and executables supplied by the NSA. To do this, the following heuristic was used:

Algorithm 1: Dispatcher Gadget Discovery Algorithm

```
Input  : Gadget G
Output: if G is valid
1  L ← Last line in G
   I ← Instruction in L
   O ← Operand in I
   if I is branch and O is modifiable then
2      R ← O
       for line in G do
3          if ( R is modified once ) and ( (CMP or TEST)
           not in line ) then
4              C ← Operand Changing R
5              for instruction in G do
6                  if C is modified then
7                      return invalid
8                  end
9              end
10             return valid
11         end
12     end
13 else
14     return invalid
15 end
```

Algorithm 1 was implemented in a Python script to find valid dispatcher gadgets from the text files supplied by the NSA. The text files contained plaintext representations of all gadgets no more than 50 instructions in length from all possible codestreams [24]. Algorithm 1 found no valid dispatcher gadgets in the 256 executables scanned.

Algorithm 1 requires that the gadget end in a branch instruction. The location branched to must be dependent on a register or some location in memory because these are both modifiable. This has to be the case, as the dispatcher gadget must change which functional gadget it sets `RIP` to. If this is true, then the memory or register used to make this jump should only be modified once, as multiple changes require control of many different registers. This quickly makes the gadget impractical.

If these conditions are met, then the operand used to modify the branching register or memory location is in-

spected. If it is modified more than once in the gadget, then the gadget is declared to be invalid. This is because it would require control of at least three seperate registers or memory locations to use the gadget, making it impractical. Also, there must be no `TEST` or `CMP` instructions in the gadget. These instructions mean that some sort of condition must be met for the gadget to be used. This makes the gadget far less flexible, as the attacker must always insure the condition is met each time the dispatcher gadget is run.

These conditions can easily be modified in the script to find less desirable, potential dispatcher gadgets. It is, however, incredibly unlikely that these gadgets would be usable. This is especially true because each functional gadget is very long, and has the potential to destroy the registers or memory locations needed for the dispatcher to work properly. Overall, the substantial length of gadgets and the protection of the shadow stack make this system significantly stronger than those which have come before it[11, 12, 28, 31].

4.1 Algorithm Results

Upon running Algorithm 1 on the text files, no dispatcher gadgets meeting the criteria were found. The code base searched was extensive, as it included all libraries and binaries present on an instrumented version of Void Linux. If the requirement that there are no `TEST` or `CMP` instructions in the gadget is removed, however, 1795 gadgets were found. This number is inflated to a degree, as many of the gadgets are part of the same codeflows. Many are found in longer or shorter versions. This is because the gadgets inspected included all possible code flows that were of length 50 or less. This change in the algorithm increases the number of possible gadgets substantially, but they are much harder to use as each requires the control of one or more additional registers. They also require that the proper state of these registers is maintained when a functional gadget is used. Given that functional gadgets will also contain many unwanted instructions for the attacker, this would be very difficult to achieve.

It is important to note that Algorithm 1 does not prove a dispatcher gadget meeting these criteria can never exist within this CFI system. It is possible to write one within source code using inline assembly. It is also possible that a usable gadget could occur based on the specific executable. Given that all libraries and executables on the Void Linux operating system were tested, this large codebase shows that it is unlikely that a useful dispatcher gadget will ever occur naturally. Furthermore, it is likely that no clean dispatcher gadgets such as the one found in Listing 6 will show up in a binary without tampering.

5. Improving the Mitigations

As shown in Section 4, JOP attacks are impractical with the proposed mitigations, and ROP is impossible because of the shadow stack. Exploits can still be made, as shown in Section 3. Therefore, the system must be improved to be totally secure.

One possible solution would be instrumentation that ensures no library functions are called which are not used by the program at runtime. This sort of instrumentation could be used on legacy systems, as binaries would simply be run through a program or an updated kernel that provides the necessary changes. Based on other forms of CFI, this would come with a high performance overhead, making it less likely to be adopted [11, 12, 30, 31]. Also, if a dangerous function like `system()` is used by the program, this mitigation would be ineffective.

Another solution would be a hardware enforced shadow hash table which saves function pointers when they are assigned. Function pointers are assigned at runtime, and generally do not need to be modified after they are assigned in most programs. Whenever a function pointer is used, its value could be checked against the values in the hash table. If the function pointer is found to be invalid, then the program faults.

This change and the shadow stack completely prevent an attacker from altering control flow by maliciously changing a function pointer, a necessary step in most code-reuse attacks [16]. This solution would be fast, as it is implemented by hardware. It would, however, not change legacy hardware in any way. Older systems would still be vulnerable. This is a significant downside that must be considered.

To avoid this downside, there could be compiler changes that place function pointers in read only memory. This technique, known as memory safety, has been used in many newly proposed protections [3, 11, 19]. It is also used in many Linux applications which can make vtables and GOT function pointers read only[39]. Overall, some of these protections can be imposed with minimal overhead if they adopt slightly weaker policies [19]. The main issue with compiler changes is the fact that legacy applications would need to be recompiled and would possibly be broken. While the overhead can be reduced, the slowdown would still be significant compared to a hardware enforced form of security.

COOP is a difficult challenge because it does not necessarily require a function pointer to be overwritten in order to work [34]. Therefore, most mitigations that are currently being developed cannot defend against it.

There is only one proposed mitigation that can defend against COOP [13]. Crane et al. propose a defense that randomizes all areas of memory, making it much more difficult for an attacker to know where dangerous functions like `system` and corruptible objects reside [13]. Furthermore, they introduce trap pages in memory that, when accessed, cause a program to fault and rerandomize itself. This technique, however, does have a counter [27]. Oikonomopoulos et al. proposed a method to get around this mitigation by allocating large chunks of memory which allow an attacker to accurately guess where other important data resides [27]. This removes the safety brought by fully randomizing memory.

The ability to map individual bytes as read-only rather than entire pages could defend function pointers and vptrs. This solution would require major changes to compilers or current hardware. The performance overhead for a solution such as this would likely be severe, so it could only be practically achieved with new hardware and updated kernels that can take advantage of this feature. It falls into the same bag as the shadow hash table and cannot secure legacy systems.

It seems as though hardware based solutions should be used in newer systems, and software solutions must be used with legacy systems. Legacy systems are still vulnerable, but instrumentation can make them much more challenging to exploit. Updated hardware can create fast, efficient programs for the future which are totally secure.

6. Conclusion

In this paper, situations were shown in which the NSA's mitigations are vulnerable. These types of exploits have been demonstrated on large, commercially used programs [16, 34]. Furthermore, JOP attacks are shown to be improbable with these mitigations in place. Finally, possible solutions to prevent the exploits that could be used against this system have been proposed and assessed.

There are many avenues available for further research. For the proposed technology, creating large, instrumented versions of programs such as Mozilla Firefox that are written in C++ would be beneficial to provide proof of concept exploits of the discussed vulnerabilities on realistic systems. Furthermore, developing a prototype for the new CPU would allow overhead due to the shadow stack and LPIs to be assessed. While JOP and COP are similar, they are not the same [5, 33]. Developing an algorithm similar to Algorithm 1 would allow the validity of a pure COP attack to be assessed.

Moving beyond the new hardware to CFI as a whole, there is much work to be done. There are practical versions of course-grained CFI on modern operating systems, namely EMET on Windows 10 [21]. While this software is an improvement upon current security, this type of loose CFI was shown to be exploitable years ago [9].

The security community does not yet have a sufficient answer for COOP [34]. Current proposals for CFI fail to consider this kind of attack and are more focused on ROP [11, 12, 28, 31]. There are randomization methods that make COOP much more difficult, but not impossible [13, 27]. COOP presents a novel challenge as it does not present any characteristics that would be easily detectable by CFI.

The future of memory corruption vulnerabilities is not all doom an gloom. Brilliant defenses such as ASLR, stack canaries, and DEP have made the task of writing exploits incredibly difficult. Currently, it takes substantial amounts of time and effort to develop the skills necessary for exploitation of large, commonly used programs. To end the current cat and mouse game plaguing this domain, new solutions need to provide guarantees which create unexploitable software while keeping overhead to a reasonable level.

8. References

[1] A. One. "Smashing the Stack For Fun And Profit." Phrack Magazine, Volume 0x07, Issue 49, File 14 of 16, 1996.

[2] Abadi, Martn, et al. "Control-flow integrity." Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005.

[3] Akritidis, Periklis, et al. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." USENIX Security Symposium. 2009.

[4] Bittau, Andrea, et al. "Hacking blind." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.

[5] Bletsch, Tyler, et al. "Jump-oriented programming: a new class of code-reuse attack." Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011.

[6] Boorstin, Julia. (2015), "The Sony Hack: One year later" http://www.cnbc.com/2015/11/24/the-sony-hack-one-year-later.html

[7] Bosman, Erik, and Herbert Bos. "Framing signals-a return to portable shellcode." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.

[8] Bryant, Randal E., and David R. Ohallaron. "x86-64 Machine-Level Programming." Retrieved May 3 (2005): 2013.

[9] Carlini, Nicholas, and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses." USENIX Security. Vol. 14. 2014.

[10] Chen, Ping, et al. "Automatic construction of jump-oriented programming shellcode (on the x86)." Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011.

[11] Chen, Xi, et al. "StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries." NDSS. 2015.

[12] Cheng, Yueqiang, et al. "ROPecker: A generic and practical approach for defending against ROP attack." (2014): 1.

[13] Crane, Stephen J., et al. "It's a TRaP: Table randomization and protection against function-reuse attacks." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.
[14] Davi, Lucas, et al. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection." USENIX Security. Vol. 14. 2014.
[15] Dragon Sector. "One-gadget RCE on Windows." https://www.yumpu.com/en/document/view/37809267/dragons-ctf/42
[16] Evans, Isaac, et al. "Control jujutsu: On the weaknesses of fine-grained control flow integrity." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.
[17] galois. "What is LandHere." http://landhere.galois.com/
[18] Gktas, Enes, et al. "Out of control: Overcoming control-flow integrity." Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014.
[19] Kuznetsov, Volodymyr, et al. "Code-Pointer Integrity." OSDI. Vol. 14. 2014.
[20] Livine, A. "Google Chrome Exploitation." 2014. http://researchcenter.paloaltonetworks.com/2014/12/google-chrome-exploitation-case-study/
[21] Microsoft. "Enhanced Mitigation Experience Toolkit (EMET) version 5.5 is now available." https://blogs.technet.microsoft.com/srd/2016/02/02/enhanced-mitigation-experience-toolkit-emet-version-5-5-is-now-available/
[22] Microsoft. "Data Execution Prevention (DEP)." https://technet.microsoft.com/en-us/library/bb457155.aspx
[23] MITRE. "CVE-2011-2439." 2011. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2439
[24] National Security Association. "A Proposed Hardware-Based Method for Stopping Known Memory Corruption Exploitation Techniques." https://github.com/iadgov/Control-Flow-Integrity
[25] NSA, "Control flow integrity." https://github.com/iadgov/Control-Flow-Integrity, 2015.
[26] Nergal. "The advanced return-into-lib(c) exploits: PaX case study." http: //phrack.org/issues/58/4.html, 2001.
[27] Oikonomopoulos, Angelos, et al. "Poking holes in information hiding." USENIX Security. 2016.
[28] Onarlioglu, Kaan, et al. "G-Free: defeating return-oriented programming through gadget-less binaries." Proceedings of the 26th Annual Computer Security Applications Conference. ACM, 2010.
[29] "one-gadget RCE in Ubuntu 16.04 libc." https://kimiyuki.net/blog/2016/09/16/one-gadget-rce-ubuntu-1604/
[30] Pappas, Vasilis, Michalis Polychronakis, and Angelos D. Keromytis. "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
[31] Pappas, Vasilis, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing." USENIX Security. Vol. 30. 2013.
[32] PaX Security. "PAX ASLR." https://pax.grsecurity.net/docs/aslr.txt
[33] Sadeghi, AliAkbar, Salman Niksefat, and Maryam Rostamipour. "Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions." Journal of Computer Virology and Hacking Techniques (2017): 1-18.
[34] Schuster, Felix, et al. "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications." Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, 2015.
[35] Shacham, Hovav. "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. ACM, 2007.
[36] Intel. Control-Flow Enforcement Technology Preview. June 2016.
[37] Snow, Kevin Z., et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.
[38] Stack Canaries. https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries
[39] Trapkit. "RELRO - A (not so well known) Memory Corruption Mitigation Technique." http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html
[40] Yu, Yang. "Write once, pwn anywhere." Black Hat USA (2014).
[41] Zhang, Mingwei, and R. Sekar. "Control Flow Integrity for COTS Binaries." Usenix Security. Vol. 13. 2013.