ROP is Still Dangerous: Breaking Modern Defenses

Nicholas Carlini David Wagner
University of California, Berkeley

Abstract

Return Oriented Programming (ROP) has become the exploitation technique of choice for modern memory-safety vulnerability attacks. Recently, there have been multiple attempts at defenses to prevent ROP attacks. In this paper, we introduce three new attack methods that break many existing ROP defenses. Then we show how to break kBouncer and ROPecker, two recent low-overhead defenses that can be applied to legacy software on existing hardware. We examine several recent ROP attacks seen in the wild and demonstrate that our techniques successfully cloak them so they are not detected by these defenses. Our attacks apply to many CFI-based defenses which we argue are weaker than previously thought. Future defenses will need to take our attacks into account.

1 Introduction

The widespread adoption of DEP, which ensures that all writable pages in memory are non-executable, has largely killed classic code injection attacks. In its place, Return Oriented Programming (ROP) has become the attack technique of choice for nearly all modern exploits of memory-safety vulnerabilities. In a ROP attack, the attacker does not inject new code; instead, the malicious computation is performed by chaining together existing sequences of instructions (called *gadgets*) [27].

In response to this, there has been a large effort to find defenses that protect against ROP attacks. Defenses fall in to two broad categories. The first category of defenses relies on recompilation to remove potential gadgets from the program binary or to enforce the Control-Flow Integrity (CFI) [4] of the binary. The other category of defenses attempts to transparently protect legacy binaries using runtime protections.

In this paper, we present three attack methods that can be combined to break many existing ROP defenses from both of these categories. Our first method breaks the conventional wisdom that it is difficult to mount attacks in a fully *call-preceded* manner, that is, where the instruction before each gadget is a `call`. Many CFI-based defenses rely upon policies similar to this. Next, we show that while most existing ROP attacks consist entirely of *short* gadgets, it is possible to mount attacks which consist of long gadgets as well. Therefore, defenses that distinguish a ROP attack from normal execution by looking for a sequence of short gadgets are not secure. Finally, we examine defenses that record a limited history of the execution state of a process. We show it is possible to effectively clear out any history kept by these defenses, rendering them ineffective.

We use these attacks to break two recent state-of-the-art runtime defenses, kBouncer [23] and ROPecker [11]. These defenses are particularly interesting because they can be deployed on existing hardware, have nearly zero performance overhead, and do not require binary rewriting. kBouncer [23] takes advantage of hardware support for recording indirect branches and examines this history at each system call in order to prevent ROP attacks from issuing any malicious syscalls. ROPecker [11] extends kBouncer in novel ways. In addition to checking for any signs of a ROP attack at each system call, ROPecker additionally checks for attacks at various points throughout program execution.

We show that both of these schemes are broken. While they may detect existing ROP attacks, we give ways of modifying a ROP attack so it will not be detected by either of these defenses. The attacks we develop in breaking these defenses are also applicable to many recent CFI-based approaches, and discuss how our work can be applied to four in particular.

This paper makes three contributions:

1. We introduce three novel ROP attacks methods that demonstrate weaknesses in multiple defenses.

2. We demonstrate these attacks on kBouncer and ROPecker, two state-of-the-art ROP defenses. We modify real-world exploits, which these defenses were shown to prevent, to bypass them.

3. Our attacks provide a baseline set of attacks that can be used to evaluate future ROP defenses.

2 Introduction to ROP Attacks

Return Oriented Programming (ROP) [27] is a generalization of return-into-libc [24] attacks where an attacker causes the program to return to arbitrary points in the program's code. This allows one to perform malicious computation without injecting any new malicious code by only controlling the program's execution flow. It has been shown that ROP can perform Turing-complete computation [30]. We provide a very brief overview of return oriented programming in this section. For a more complete introduction, we refer the reader to [7, 25, 27].

A ROP exploit consists of multiple *gadgets* that are chained together. Each gadget performs some small computation, such as loading a value from memory into a register or adding two registers. In a ROP attack, the attacker finds gadgets within the original program text and causes them to be executed in sequence to perform a task other than what was intended.

Gadget chaining is achieved by influencing indirect jumps executed by the program. Each gadget begins with some useful instructions (e.g., `mov rax, rbx`) and ends with an indirect jump (e.g., `ret` or `jmp *rcx`). The attacker chains gadgets together by controlling the target of a gadget's indirect jump to point to the beginning of the next gadget in the sequence. In a classic ROP attack, gadgets end with the `ret` instruction and the attacker chains gadgets by writing appropriate values over the stack.

Many ROP attacks use *unintended* instruction sequences. Because x86 instructions are variable-width, it is possible that a potentially useful gadget sequence exists when starting at an offset that was not intended to be the beginning of an instruction. Our attacks do not rely on unintended instructions.

In Figure 1, we give an example ROP exploit that adds 0x32400 to the value stored at address 0x4a304120. This exploit begins by initializing two registers. It then reads the value stored at address `eax`, stores it into `eax`, adds `ebx` to `eax`, and stores this value back into memory.

Address Space Layout Randomization (ASLR). One common defense for ROP attacks is ASLR which works by randomly moving the segments of a program (including the text segment) around in memory, preventing the attacker from predicting the address of useful gadgets. Despite ASLR, ROP attacks are still common in the wild for two reasons. First, if even a single module has ASLR disabled, a ROP attack may be formed around only the code in that module. Second, an attacker may use an *information disclosure vulnerability* to de-randomize some module [29].

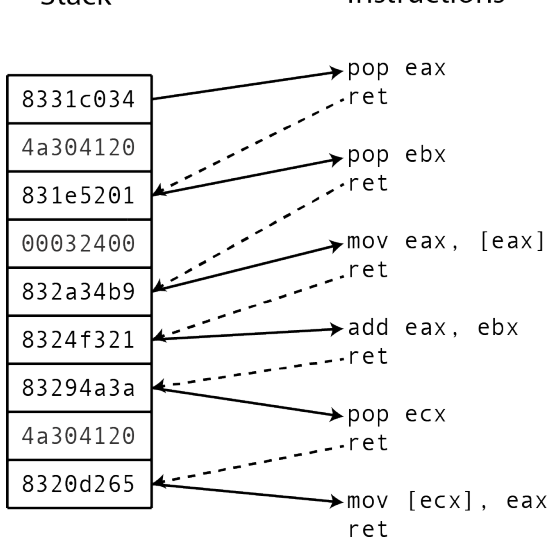

Figure 1: An example ROP exploit which adds the constant 0x32400 to the word at address 0x4a304120. At the left is the stack of the process with the addresses of the gadgets and the values to initialize the registers. At right are the instructions at those addresses.

3 Our Three Attack Primitives

We have identified three building blocks that are useful in attacking ROP defenses:

- *Call-Preceded ROP.* Normally, in a well-structured program, every `ret` instruction returns back to an instruction that immediately follows a corresponding `call`. ROP attacks deviate from this pattern. Therefore, many ROP defenses ensure that every `ret` instruction always targets an instruction that immediately follows some `call`. Our attack demonstrates that this policy is not sufficient: ROP attacks are still possible even when returns are restricted in this way.

- *Evasion Attacks.* It is common for defenses that monitor program execution at runtime to have a method of classifying execution as either "normal execution" or "gadget". Evasion attacks involve using gadgets that the defense classifies as "normal."

- *History Flushing.* Some defenses maintain only a limited amount of history about execution and inspect this history periodically. We can bypass defenses with this property by flushing the true history (cleansing the history of all signs of the ROP attack) and then presenting a new, fake view of history that the defense will not classify as an attack.

Each of these three attack primitives bypasses a common defense mechanism. This section gives more detail about each of these three primitives. We then combine them in different ways to mount our full attacks on kBouncer [23] and ROPecker [11] in the following sections.

3.1 Call-Preceded ROP

The call-preceded policy. We say that an instruction is *call-preceded* if the instruction immediately preceding it is a `call` instruction. Many ROP defenses [6, 23, 32, 34] apply the following policy: any time a `ret` instruction is executed, its target must be a call-preceded instruction.

This policy seems helpful for defending against ROP attacks. In well-structured programs, calls and returns usually come in pairs. Any address that is returned to was almost always pushed by a `call` instruction previously. In a ROP attack, gadgets use the `ret` instruction to chain gadgets together, so this policy dramatically limits the space of candidate addresses where gadgets can be chosen from. For instance, one evaluation found that only 6% of gadgets are call-preceded [23]. Thus, one might intuitively expect the call-preceded policy to significantly increase the difficulty of mounting a ROP attack.

Using only call-preceded gadgets. Despite this intuition, we find that it is possible to mount ROP attacks in a fully call-preceded manner, where all gadgets start at a call-preceded address. The key idea is we allow gadgets to be more complex. This increases the space of candidate gadgets enough to find many call-preceded gadgets. By allowing our gadgets to be long and contain direct jumps or even conditional jumps, we find many more useful gadgets. In our experiments (see § 8.2), 70KB of binary code was sufficient to mount fully call-preceded ROP attacks.

3.2 Evasion Attacks

Classification-based defenses. Other ROP defenses work by monitoring the runtime behavior of a process and try to detect ROP attacks by classifying segments of execution as either "gadget" or "non-gadget", using some signature that is intended to characterize attributes of ROP gadgets. One of the most common approaches used to classify execution, as used in [11, 23], uses a length-based classifier. Existing ROP attacks tend to consist of long sequences of short gadgets, and so these defenses use this as their heuristic to classify gadgets.

These defenses separate the execution trace into segments of ordinary instructions, separated by indirect instructions (e.g., returns, indirect jumps). A length-based defense classifies each segment as gadget or non-gadget by examining its length: a short segment is classified as a gadget and a long segment as a non-gadget. If the defense observes too many short segments within some window, it reports a ROP attack.

Using gadgets that look like benign execution. A powerful attack on such defenses is to look for instruction sequences that would be classified by the defense as a non-gadget, but that perform some useful computation. These can then be used as stealthy gadgets in a ROP attack.

Length-based classifiers are particularly easy to evade. A simple attack is to use long gadgets, since these will be incorrectly classified by the defense as non-gadget. We demonstrate that it is possible to mount a ROP attack that contains a mixture of both short and long gadgets, thus evading many published detectors.

More generally, one could imagine future ROP defenses that rely on other heuristics for distinguishing ROP attacks from normal program execution. An *evasion attack* is one that will be classified by the defense as normal, but in reality allows the attacker to mount a ROP attack.

3.3 History Flushing

History inspection defenses. There are many runtime defenses that inspect program execution at different points throughout its execution. Typically, these defenses keep only a limited amount of history about the program's execution, and so must decide whether an attack is occurring or not based upon information saved in the recent past. Usually, performance considerations rule out constantly monitoring all execution, so this inspection process is only invoked at certain points (e.g., when the application issues a system call).

Using gadgets to hide history. Such defenses can be fooled by preventing them from seeing any evidence of a ROP attack. We perform the ROP attack when they are not watching, periodically performing enough innocuous actions to wipe the history clean of any evidence of the past ROP attack before the defender's inspection process is invoked. While the defender is running, we do not attempt to make progress towards our attack goal. Instead, we insert effective no-op instructions so that the defender does not see any evidence of attack.

Though similar, this attack is different from an evasion attack. An evasion attack attempts to make progress in the attack while being *continuously* monitored by the defender. In a history flushing attack, there is a period of time when the defender is not running, when we make forward progress. Before the defender runs, we clear out this history so it is not visible to the defender, but do not attempt to make forward progress while the defender is watching. After the defender has made its observation, we continue with our attack.

For instance, kBouncer uses the Last Branch Record, a

hardware feature that records the 16 most recent indirect jumps. Our history-flushing attack on kBouncer performs the bulk of the ROP attack, then performs 16 innocuous indirect jumps to remove the evidence of the ROP attack from the Last Branch Record. As we show (§ 8.3), this prevents kBouncer from detecting the ROP attack.

4 Attack Goal & Threat Model

Attack Goal. The goal of each of our attacks, without loss of generality, is to issue a single syscall. It is usually enough to issue a `mprotect` (on Linux) or `VirtualProtect` (on Windows) system call to make a page in memory both writable and executable; after that, exploitation is trivial.[1]

This is not the only possible goal an attacker may have. There are other methods of attack that do not involve issuing system calls [10]. We do not consider them in this work, although our results suggest these attacks are equally possible, and in some cases even trivial.

Threat Model. At a minimum, we assume that an attacker has a known exploit that allows control of the instruction pointer in the future. A stack overflow is sufficient; a heap overflow that allows an arbitrary memory write to a function pointer is also sufficient; as is directly overwriting other function pointers. We assume the attacker knows that the defense is present and knows how it works. We assume that DEP is enabled, so no page is both writable and executable. We focus on the case where the program contains at least one library whose executable region has not been randomized with ASLR, or where all modules have ASLR enabled but there exists a memory disclosure vulnerability, as this is the situation that modern ROP attacks typically exploit.

We also assume that there exists some way of running arbitrary code if the new defenses were not present. We do not claim to create attacks that allow running arbitrary code in all situations; we only hope to show that if it is possible to mount a ROP attack when the defense is not present, then it is possible when it is present.

5 Defeating kBouncer

5.1 Overview of kBouncer

Pappas et al. introduced kBouncer [23], a scheme that uses indirect branch tracing to detect ROP attacks. At a high level, kBouncer periodically pauses execution of the program, inspects recent execution history, and then either allows the process to proceed or kills it.

[1] Alternatively, if we can execute the `execve` syscall, we can spawn a second process running an arbitrary program.

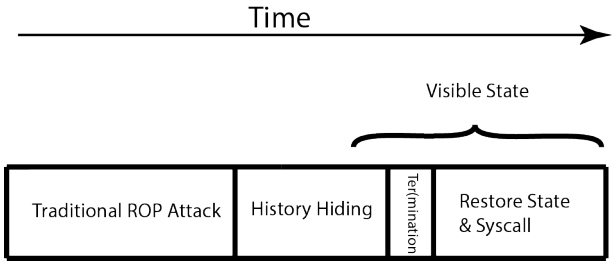

Figure 2: Overview of our history hiding attack on kBouncer. We mount a traditional ROP attack, insert a number of innocuous gadgets to hide this from kBouncer, and finally restore registers and issue the desired syscall.

kBouncer uses the Last Branch Record (LBR), a feature of modern Intel CPUs, to inspect the last 16 indirect branches taken each time the program invokes a system call. kBouncer checks two properties of the history stored in the LBR. First, it verifies that all `ret` instructions in the LBR returned to a call-preceded address. Second, if the eight most recent indirect branches are all gadget-like, the process is killed. kBouncer defines a sequence of instructions as *gadget-like* if there exists a flow of execution from the first instruction executed to any indirect branch in under 20 instructions.[2] kBouncer is very efficient: it only needs to check the LBR during system calls and only checks 16 different entries in the LBR.

5.2 History Hiding Attack

5.2.1 Attack Overview

We dub our first attack on kBouncer the *history hiding attack* (see Fig. 2). At the core of kBouncer is the assumption that an attack can be detected by inspecting the state of the process at the syscall interface, after the attacker has already gained control of the system for a potentially unbounded period of time. After mounting a traditional ROP attack to prepare the state of memory (and possibly defeat ASLR, if required), we use a history flushing attack to clear evidence of the attack from the LBR. Finally, we use an evasion attack and a few carefully-chosen gadgets to issue the syscall.

We call a process state *valid* if kBouncer's inspection method will not detect an attack when run from that state. A state is valid if all of the entries in the LBR whose source is a `ret` instruction have a call-preceded destination, and if at least one of the last eight entries has more

[2] kBouncer cannot observe the actual path of execution taken during a sequence of instructions between two indirect jumps, so it cannot count the number of instructions actually executed between two indirect jumps. It can only observe the beginning and end of that sequence. For this reason, kBouncer conservatively treats a sequence as gadget-like if it starts with an instruction that can reach an indirect jump in less than 20 instructions.

than 20 instructions between source and the nearest indirect branch. We show that it is easy to return to a valid state while simultaneously maintaining control of the process. The steps of the history hiding attack are as follows:

Initial exploitation. Initially, we mount a traditional ROP attack in whichever way is easiest. We ignore the fact that kBouncer is running and use any gadgets we would like, call-preceded or not. We then prepare memory so we are ready to make the syscall, but we do not invoke it yet.

Hide the history. At this point in our exploit, we are ready to make the syscall, but if we were to actually issue it, kBouncer would detect an attack. To fix this, we must bring the process into a valid state without losing our progress from the prior step. To do this we use the history-flushing primitive discussed previously. As a side effect of using the flushing primitive, the registers may be clobbered, but important memory locations will remain unchanged.

Restore registers and issue the system call. After bringing the process into a valid state, we restore the registers to their desired values while maintaining a valid state. Then, we issue the system call. This is via an evasion attack: because the task is relatively simple, it can be accomplished with fewer than 8 call-preceded gadgets.

```
(1)
   pop  ebp
   ret

(2)
   jmp  A
   ...
A: mov  eax,3
   ret

(3)
   cmp  eax,6
   jbe  B
   ...
B: ret

(4)
   xor  eax,eax
   ret

(5)
   mov  [eax],0
   ret
```

(a) Flushing Gadgets

```
      add  [esp+17Ch],ebx
      mov  ebx,[esp+17Ch]
      sub  ebx,ebp
      jmp  A
      ...
A:    add  [esp+64h],ebx
      jmp  B
      ...
B:    mov  esi,[esp+1C0h]
      lea  eax,[esi*8-4]
      sub  eax,[esp+64]
      and  eax,7h
      mov  edi,[esp+64]
      lea  eax,[edi+eax+4]
      shr  eax,3
      cmp  eax,esi
      jbe  C
      ...
C:    mov  eax,[esp+1C0h]
      add  esp,19Ch
      pop  ebx
      pop  esi
      pop  edi
      pop  ebp
      ret
```

(b) Termination Gadget

Figure 3: Examples of the two types of gadgets used by our history-hiding attack on kBouncer. A flushing gadget flushes the contents of the LBR. A termination gadget brings the system into a valid state.

5.2.2 Initial Exploitation

This step prepares memory to make it as easy as possible to issue the syscall in as few gadgets as possible after the history has been flushed. In particular, we prepare all of the arguments for the system call and save them in some easily recoverable location. We make no restrictions on the methods the attacker may use during this step of our attack. Because we are going to hide our history, kBouncer will not observe anything performed in this step. Since ROP gadgets are Turing-complete, we are able to perform arbitrary computation during this phase, so this step is straightforward to implement.

5.2.3 Hiding the History

Hiding history through LBR flushing. We use a history-flushing primitive, built from two gadgets (Fig. 3), to remove all traces of our attack from the LBR:

1. A short *flushing* gadget: a simple call-preceded gadget that performs a `ret`, and ideally does not modify many registers.

2. A long *termination* gadget: a call-preceded gadget that is long enough for kBouncer to not classify it as a gadget: there must be at least 20 instructions along every possible control path from the start of this gadget to any indirect branch.

We use these two gadgets as follows. First, we repeatedly use the flushing gadget to completely clear the contents of the LBR until it only contains the flushing gadget repeated 16 times. Though the LBR has been flushed and contains no history of the previous ROP attack, the state is still not valid. If kBouncer were to be invoked at this point, every entry in the LBR would be classified as a gadget by kBouncer and an attack would be detected.

We now use the termination gadget. The purpose of this gadget is to bring the LBR into a valid state by making at least one of the last eight entries in the LBR have length greater than 20. That is, the termination gadget is used to *terminate* kBouncer's backwards search for gadget-like sequences. We make no assumptions about the register state after the termination gadget is executed: the only requirement is that after we use it, we still have control of instruction flow.

```
4a833dd4        dec     ecx
4a833dd5        fmul    [4A88BBC8h]
4a833ddb        jne     4A833DD4
```

Figure 4: An example of a context switch gadget found in `icucnv36.dll`.

Note that during the first step where the attacker prepares memory, the attacker may perform arbitrarily complex calculations. This may make it possible to initialize registers and memory so that executing the flushing gadgets and then the termination gadget results in exactly the desired state to issue the syscall. However, this is not always possible. For example, the termination gadget may set `eax` to 0, but issuing the syscall may require `eax` to be 7. Our attack handles this situation by later restoring register state (described below).

Because the termination gadget is over twenty instructions long and might contain conditional branches, it is sometimes necessary to initialize registers and memory to meet the preconditions for successful execution of the termination gadget. First, we need to ensure that any conditional branches in the termination gadget will be followed in a specific manner. Second, memory reads and writes must not fault and crash the process. This is often as easy as initializing registers to specific values before using the termination gadget. We have found that termination gadgets are very common, and that it is often easy to find termination gadgets that perform only a few conditional branches and memory reads and writes (see § 8.3.1).

History hiding by itself does not defeat kBouncer, but it simplifies the attacker's job from expressing the entire attack using call-preceded gadgets to expressing only the final step of the attack using call-preceded gadgets.

Hiding history through context switching. We also found an alternative way to flush history. The LBR is shared across all user-space processes. This lets us flush the LBR using a single gadget, the *context switch* gadget. A context switch gadget is one that will run for many seconds and will not contain any indirect branches. The simplest way to find such a gadget is to look for loops that perform a very limited computation using only registers, see Fig. 4 for one such example.

To flush the LBR, we call the context switch gadget once. Due to the number of cycles this gadget takes to execute, it is almost certain that there will be several context switches to other user threads during its execution. When this happens, the other thread will write its own entries to the LBR, flushing all history of our prior attack. Eventually, when our context switch gadget finishes, the LBR will be in a valid state as long as the other process was not under attack, as the LBR is now full of innocuous entries from the other process.

Future hardware could save and restore the LBR on context switches, which would prevent this method of history flushing. Therefore, we did not use this approach in our case studies (§ 8); instead, we used flushing and termination gadgets, which would suffice to hide history even if the LBR was saved and restored on each context switch.

5.2.4 Restoring Registers with Returns

We must now restore the registers to their desired values in order for the syscall to proceed. This is by far the simplest step and can be usually be accomplished with a few gadgets that pop register values off the stack. kBouncer will be able to observe each gadget we use, so each one must be call-preceded and we must use fewer than eight.

This step is often very easy because of the x86 calling convention: the procedure being called must restore almost all of the registers, so procedures tend to begin by pushing all of the registers onto the stack and end by popping those values off to restore them. This allows us to find a gadget that pops all the registers off the stack and then returns. Usually, we can find all the (call-preceded) gadgets we need in this way.

5.2.5 Restoring Registers without Returns

There are other ways to restore register state. We now discuss four alternative methods. The first two are existing techniques that can be applied here, but in our experience are difficult to apply in practice due to the fact that we must use fewer than eight gadgets. We have found the later two techniques more applicable in practice.

ROP without return instructions. Checkoway et al. found it is possible to mount a ROP attack by looking for a `pop` followed by an indirect jump (e.g., `pop edx; jmp *edx`) [8]. This instruction sequence is functionally identical to a `ret`, and so can simply be used in its place. However, these sequences are less common.

Jump Oriented Programming (JOP). JOP attacks use register-indirect jumps to chain gadgets together. Unfortunately, each useful gadget must be followed by a dispatcher gadget, which is used for chaining. Since we must restore register state with at most eight gadgets, if we want to use JOP, we are limited to four useful JOP gadgets.

Using Non-Call-Preceded Gadgets. Occasionally, it may be easier to use non-call-preceded gadgets. We can invoke a non-call-preceded gadget using a *reflector* gadget. A reflector gadget is a call-preceded gadget that ends in a register-indirect jump; it can be used to jump to any gadget we like, call-preceded or not. This is because kBouncer imposes no constraints on indirect jumps. Our

experience is that this trick is rarely needed in practice, but sometimes it makes constructing the attack easier.

Call Oriented Programming (COP). We have found an alternate method of mounting a ROP-like attacks without using `ret` instructions. We call our approach Call-Oriented Programming (COP). Instead of using gadgets that end in returns, we use gadgets that end with indirect calls. This may at first seem trivially similar to jump-oriented programming, but there is one important distinction: indirect calls are usually memory-indirect (the location to which control is transferred is determined by a value in memory, not directly by the value of a register). As a result, COP attacks do not require a dispatcher gadget. In a COP attack, gadgets are chained together by pointing the memory-indirect locations to the next gadget in sequence. The initialization of these memory locations can be done *in advance*.

This allows our attack to set up these memory locations before the history hiding, then restore register state using COP gadgets. As long as fewer than eight COP gadgets are used, kBouncer will detect no attack. When mounting a COP attack, it is trivial to directly issue the desired system call as well: the final gadget in the sequence will point to the system call to be issued.

We have found that memory-indirect calls, and in particular COP gadgets, are common. They are even more common than call-preceded gadgets that end in a `ret`. There are two reasons why this is the case. First, with dynamically linked libraries, all calls to functions outside of the current module are indirect calls, because the function location is not known in advance. Second, most object-oriented code relies on memory-indirect calls (e.g., the vtable in C++).

COP attacks do not eliminate the need for `ret`-based gadgets. Initializing a COP attack is much more difficult: the attacker must have control of program flow, must overwrite specific indirect-call locations, and must control the stack. This usually is not possible with a single exploit. Therefore, it is natural to combine a ROP attack (for initial setup) with a COP attack (for restoring registers).

5.2.6 Issuing the System Call

The final step of our attack is to issue the desired syscall. We usually accomplish this by calling the appropriate `libc` or `kernel32` wrapper function.

There is one complication. We cannot simply return directly to the beginning of the desired function (e.g., `mprotect`, `VirtualProtect`) as a normal ROP attack would. When kBouncer is in place, this is not possible: the attack would fail because the start of this function is not call-preceded. We have found three different ways to call a function without directly returning to it.

```
    call [7C37A094]
A:  mov eax,[_osplatform]
    jmp B
    ...
B:  dec eax
    neg eax
    sbb eax,eax
    and eax,103
    lea ecx,[ebp-0Ch]
    push ecx
    inc eax
    push eax
    push [EBP-8]
    push [EBP-4]
    call [VirtualProtect]
```

Figure 5: A call-preceded call to `VirtualProtect` in `msvcr71.dll`. The attacker can return directly to `A`.

1. We can use a *reflector gadget*: a call-preceded gadget that ends with a register-indirect jump. This allows us to simply set a register to point to the function we wish to call and then return to the reflector gadget. This is the simplest approach if a reflector gadget can be found.

2. It is still possible to exploit the desired function even if no reflector gadgets are available. This is achieved by finding an call to the desired function somewhere in the program's code and looking backwards in the instruction sequence for a preceding `call`. Fig. 5 shows an example where the `msvcr71.dll` binary directly calls `VirtualProtect`.

3. It is sometimes possible to return into the middle of a desired function, right after a `call` instruction. For example, `execv()` launches a shell with a string and an array of arguments (Fig. 6). If we initially initialize `rax` to contain a valid environment pointer, we can call `execv` by returning directly to `<execv+18>`, which is call-preceded.

Any of these can be used to complete our attack.

5.3 Evasion Attack

Our history hiding attack breaks kBouncer by taking advantage of its limited history. If kBouncer were extended to have a complete view of history, would it become more effective? We show that, even if the LBR were of infinite size, kBouncer could still be broken by an evasion attack.

Our attack is similar to the history hiding attack (§ 5.2), except that the initial preparation phase is mounted using only call-preceded gadgets. This eliminates the need

```
<excve>:
    push   rbp
    mov    rbp,rsp
    push   r14
    push   rbx
    mov    r14,rsi
    mov    rbx,rdi
    call   _NSGetEnviron
    mov    rdx,[rax]
    mov    rdi,rbx
    mov    rsi,r14
    call   execve
    mov    eax,-1
    pop    rbx
    pop    r14
    pop    rbp
    ret
```

Figure 6: Disassembly of the `execv` function in libc on our system. The call to `_NSGetEnviron` allows a call-preceded return directly into this function.

for a flushing gadget, the only piece that an infinite-LBR kBouncer would preclude. Therefore, our attack consists of a (call-preceded) setup, a (call-preceded) termination gadget, followed by (call-preceded) register restoration and syscall.

This yields a successful evasion attack on kBouncer. By using only call-preceded gadgets and by breaking up the chain of short gadgets with a long termination gadget, kBouncer can see the entire attack but still will not recognize it as an attack. Our experiments show that if over 70KB of program text is available, then there are enough call-preceded gadgets that this attack is possible (see § 8.2).

6 Defeating ROPecker

6.1 Overview of ROPecker

ROPecker [11] is a ROP defense that builds on ideas found in kBouncer. ROPecker differs from kBouncer by running its inspection method more frequently and inspecting the program state more thoroughly at the time of inspection. The actual policy it enforces is very similar to the kBouncer policy.

In ROPecker, only a few pages are ever marked executable at one time. We call these pages the *executable set*. Whenever a page not in the executable set is executed, a page fault is generated and ROPecker pauses process execution to check for an attack. If ROPecker does not detect an attack, it marks the new page as executable, marks the least recently executed page as non-executable, and resumes the process. ROPecker also runs its detector whenever the process invokes a syscall as kBouncer does.

ROPecker's detector is more sophisticated than kBouncer's in that it looks at both the recent past and projects forward into the near future. Similar to kBouncer, ROPecker classifies the current state as an attack if there is a long chain of *gadget-like* sequences in the LBR (the recent past). In addition, ROPecker attempts to emulate what will happen in the near future once the process is resumed. It counts the number of gadget-like sequences that are about to execute. If the sum of the number of gadgets found in the LBR and the number of gadgets looking forward exceeds some threshold, ROPecker classifies this as an attack.

ROPecker's emulation works by disassembling the instruction stream from the instruction that is about to execute when the page fault occurs. If there is a short sequence of instructions that leads to an indirect jump, ROPecker classifies this as a potential gadget. ROPecker will then emulate the effects of each of the instructions leading to the indirect jump in order to compute where this jump will go. ROPecker follows this indirect jump and starts disassembling again. When it reaches an instruction where there is not a short sequence of instructions leading to an indirect jump, it stops the search. ROPecker then counts the number of indirect jumps followed, and classifies each of those as gadgets.

ROPecker verifies that from the current execution point there are not 11 gadget-like sequences of instructions.[3] ROPecker classifies an instruction sequence as a gadget if it contains six or fewer instructions ending in an indirect branch, with no direct or conditional branches along the way.

6.2 The Repeated History Hiding Attack

6.2.1 Attack Overview

We show how to break ROPecker using a *repeated history hiding* attack. This attack repeatedly invokes the history-hiding primitive, introduced in § 3.3, just before ROPecker's detector is about to execute. We again define a state to be *valid* if the inspection method will not detect an attack. The state must be valid at two points in time: whenever a new page is loaded in to the executable set and whenever a syscall is executed.

Our attack alternates between three phases, as depicted in Fig. 7. The *loading phase* loads useful pages into the executable set. The *attack phase* invokes gadgets on these pages. The *flushing phase* mounts the history hiding attack from § 5.2 using only gadgets from the pages that are

[3] The ROPecker paper does not pick a specific parameter for the maximum number of gadgets that may execute consecutively. It suggests this number is chosen between 11 and 16, so we conservatively pick 11. Our attacks are made easier if a larger number is chosen.

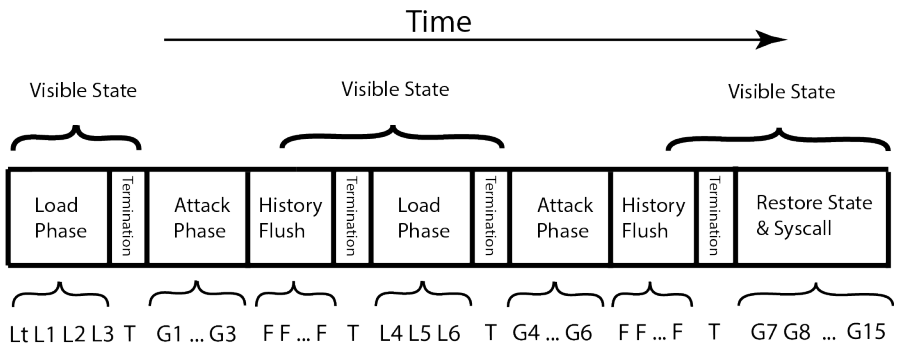

Figure 7: An overview of the repeated history hiding attack on ROPecker. `Ln` gadgets load page `n`. `Lt` loads the termination gadget. `Gn` invokes a gadget on page `n`. `F` is a short flushing gadget, and `T` is a long termination gadget.

in the executable set. We may need to execute each of these three phases multiple times to achieve our goal. We conclude with one final step which actually issues the desired syscall after restoring the required state. Because we use only gadgets in the executable set during each attack phase, ROPecker's detector will execute only when new pages are loaded, which allows us to reason about what will be visible to ROPecker.

6.2.2 Attack Phases

Initialization. Prior to our attack, we insert a termination gadget, which will stop ROPecker from looking further back in the LBR. This long termination gadget is identical to the one used in the kBouncer attacks. This ensures that when ROPecker next runs, it will not count any functions on the call stack prior to initialization as gadgets.

Loading Phase. We load useful pages into the executable set by invoking a page load gadget on each page we want added to the executable set. A *page load gadget* is any call-preceded gadget on that page, which has two properties: first, it must leave the attacker with control of the instruction flow; and second, it must not crash the process. These two requirements are not difficult to meet: any useful gadget is also a page load gadget. The ROPecker detector will run immediately before each page load gadget is invoked. After invoking each set of page load gadgets we call the termination gadget to prevent the detector from looking forward any farther into the future.

ROPecker will not detect an attack because each sequence of page load gadgets is immediately preceded and followed by a termination gadget. When a page fault occurs, ROPecker will count the number of visible gadgets looking backwards in the LBR and forwards as far as it can see. Looking backwards will stop at preceding termination gadget, and looking forward will stop at the subsequent termination gadget. Thus, ROPecker will count the number of page load gadgets. By limiting the number of consecutive page load gadgets, the attacker can evade detection during this phase.

Attack Phase. Now that the useful pages have been loaded, we can use any gadgets on these pages to mount an attack, ignoring any defense which may be running. As long as we use only gadgets on these pages, the defense will never trigger.

Recall that these three phases are repeatedly executed, so no one attack phase needs to perform the entire attack. Instead, the attack can be distributed among multiple attack phases, making each one simpler.

History Hiding. After invoking gadgets on these pages, we now use the history flushing primitive before the detection method next runs. We use the same method we applied against kBouncer to clear the LBR. In particular, we invoke a short flushing gadget enough times to fill the LBR with innocuous entries, then invoke the long termination gadget (which was loaded previously). When the ROPecker detector next runs, it will see no attack prior to this point in time.

6.2.3 Segmenting the Attack Payload

When mounting this attack, we must carefully pick which tasks to perform during each attack step. Because the flushing and termination gadgets clobber some register state between each attack step, it is important to pick small independent operations for each step of the attack.

For any given attack, it may not be possible to modify it to work as an attack which bypasses ROPecker. Instead, attacks must be formed with ROPecker in mind. Each step in the attack must be constructed to use only a limited number of gadgets, so that its work can be saved before loading in a new set of gadgets.

Often, we start by computing the address of the desired `libc` function we wish to call (e.g., `mprotect`) either by adding a constant to the address of some other function

in `libc`, or by loading it directly. We store the result in memory. In the next attack step, we compute the address of the page we wish to mark as executable (typically on the stack). We continue in this way, computing any other needed constants in separate attack steps. We then restore register values and call `mprotect` on the desired page. Finally we can execute a traditional payload with data we have written to this page.

6.2.4 Selecting Pages to Load

Since the executable set can contain only a few pages at one time, we must choose these pages with care. The naive approach is to select each page to load for one useful gadget on that page, and call each gadget exactly once. We have found that this simple method works well in practice in most cases. Because the flushing and termination gadgets may clobber a few registers, we may need reserve one or two of those gadgets to load and save registers to memory, so that a task can be partially completed in one attack step.

A more advanced method is to pick pages that contain multiple gadgets. In our evaluation, we found that in practice there tend to be many "useful" gadgets on the average page. Thus, by selecting the pages carefully, we can find pages with enough useful gadgets. This is enough that we can attack ROPecker even when the size of its executable set is limited to just one or two pages.

6.2.5 Issuing the Syscall

Once we have executed sufficient load/attack phases to set up the state of the process, we append one final step to actually issue the desired syscall. This step is not executed multiple times: it is done only once at the very end.

During this step, we flush history, invoke the termination gadget, and then issue the syscall using one of the three methods from § 5.2.6. We perform this step using at most 10 gadget invocations so that ROPecker will not detect an attack when it examines the LBR at the syscall.

Conveniently, it is possible to use any gadget in the entire binary during this step, even if it is not contained within the executable set. No page loading gadgets are needed. This works because there will be at most 10 gadgets between the termination gadget and the syscall. Thus, even though ROPecker's detector may run during this step (if we use a gadget that's not in the executable set), its count of the number of gadgets will be below 11, the threshold for detecting an attack.

Note that, in particular, an attack which requires fewer than ten gadgets to execute can skip the load/attack phases and directly issue the syscall in this way.

6.3 The Evasion Attack

We now present the ROPecker evasion attack, an alternate attack that would break ROPecker even if the size of the executable set were reduced to just *one* page. As a side benefit, in our experience the evasion attack makes it easier to automate attacks in practice than the repeated history hiding attack of § 6.2.

At a high level, the idea is that we will let ROPecker inspect the execution of our attack at arbitrary points in time. We ensure that no matter when its detector runs, it will never detect an attack. We achieve this through an evasion attack similar to the one presented on kBouncer (§ 5.3).

The ROPecker evasion attack works by inserting a termination gadget in between every ten useful gadgets. When the detector runs, it will check forward and backward to count the number of gadgets in use; there will be fewer than 11 gadgets, the threshold for detection, so ROPecker will not detect the attack.

The authors of ROPecker note that this attack may be possible in § VII(b) of their paper [11]. They propose a mitigation for such an attack. We show that even their mitigation is broken.

The ROPecker mitigation. ROPecker detects an attack if there are more than ten consecutive gadgets. The extended version of ROPecker records how many gadgets existed in previous runs of the detector. It detects an attack if the number of gadgets which executed in the last T runs is larger than some threshold. While it is possible for there to be 10 sequential gadget-like returns in benign program execution, it is unlikely for there to be 10 sequential gadget-like returns *T times in a row*.

Conceptually, this is analogous to running the detection mechanism both forwards and backwards, allowing up to $T-1$ long gadgets before stopping the search. An attack is detected if the number of gadgets found by this extended search is greater than some threshold.

This defense *does not* help against our repeated history hiding attack. In that attack, ROPecker only ever sees as many gadgets as pages that are being loaded. This constant is usually very small (e.g., two or four). The ROPecker authors observed that benign execution does occasionally execute four sequential gadget-like chains (with frequency 0.58%). This frequency is large enough that signaling an attack if there are four gadgets repeated three times would cause too many false positives.

Breaking the mitigation. The extended version of ROPecker can be broken by a simple modification of our evasion attack: instead of invoking the termination gadget once, invoke it T times in a row. We alternate making one step of useful progress (with ten useful gadgets) with invoking the termination gadget T times. This prevents

ROPecker from detecting consecutive long chains of gadgets. Instead, it sees a long chain followed by several short chains, which will not trigger the defense.

Practicality. One might wonder whether evasion attacks are practical. If, between every ten useful operations, we must potentially destroy our progress, can we achieve any useful computation?

We found it is still possible to perform useful tasks even when inserting a termination gadget (or, potentially multiple termination gadgets) in between every ten useful gadget (see § 8.3). We save register state to memory before each termination gadget and restore it afterwards. It is only necessary to save and restore registers that are both clobbered by the termination gadget and used by the rest of our attack. In our experience, it is often possible to find termination gadgets that only clobber one or two registers. This allows for many gadgets that make forward progress, with a few dedicated to saving and restoring state.

6.4 Attack Comparison

These two attacks are useful in different circumstances. The most important difference is when the detection mechanism runs. In repeated history hiding, the detection only ever runs after a history flush, and so the defender can never even see what the attacker is doing. In the evasion attack, the defender is continuously monitoring the attack progress. This leads to the key distinction between the two attacks. In repeated history hiding, we have a very limited set of gadgets, but may use them an unbounded number of times before flushing. In the evasion attack, we have all of the gadgets in the program available to us, but must flush every ten gadgets.

7 Fixable Attacks on ROPecker

We now discuss several ways in which ROPecker is broken that our attack does not rely on. That is, the attacks discussed in the previous sections work *even if we improve ROPecker's detection mechanisms* to prevent each of the following specific attacks. We believe these modifications are possible, and it is only the engineering difficulties of obtaining a low overhead that explains why they are not currently implemented. Because of this, we do not base our previous attack on these fixable implementation issues.

Gadget definition does not allow any branches. ROPecker's definition of a gadget is overly specific and does not allow gadgets to contain either direct or conditional branches. In comparison, we have found that kBouncer's definition of a gadget is strong: it is difficult to find gadgets of length twenty or more that perform useful computation.

ROPecker's choice to not follow any direct or conditional branches is a flaw that, while allowing for a more efficient implementation, makes exploitation nearly trivial. This decision allows an attacker to flush the LBR, and to stop the forward-inspection algorithm, with a no-op-like gadget that jumps directly to a return instruction. This form of gadget is pervasive in program binaries and allows for a much simpler termination gadget that does not clobber any register state.

In fact, when evaluating the practicality of our attacks on kBouncer *before becoming aware of ROPecker*, nearly all of our exploits contained at least one *useful* gadget that would not be classified as a gadget by ROPecker's definition.

Gadget chain threshold is too short. ROPecker's choice to define gadgets as being a sequence of six or fewer instructions makes it nearly trivial to find gadgets that have a predictable behavior while still being classified as a non-gadget by ROPecker. For example, on 64-bit systems, the gadget consisting of popping off registers `r10` through `r15` followed by a `ret` is seven instructions long: not only is this a useful gadget, it is very common. ROPecker's failure to recognize it as a gadget is a serious limitation of ROPecker.

The set of risky system calls is not complete. ROPecker's set of *risky system calls* is too limited and needs to be updated to more closely match those used in kBouncer. Because ROPecker is designed for Linux and kBouncer for Windows, we cannot simply replace one with the other. However, other than performance reasons, there is no reason to not defend all system calls.

8 Evaluation

The attacks discussed in the previous sections are practical. We evaluate these attacks by modify real-world exploits, as well as by demonstrating that only 70KB of code is needed to mount purely call-preceded attacks.

8.1 Our Tool

We built a tool to assist our efforts in finding attacks on real-world exploits. It does not automatically break either of these two defenses, but assists in finding useful gadgets. We wrote our tool as a 1K line Python program. It takes as input a disassembled object file (from `objdump`), and therefore only inspects *intended* instruction sequences: even though there may be unintended instruction sequences which are call-preceded, we ignore these.[4]

[4]Even though ROPecker does not enforce gadgets are call-preceded, we still use this tool to evaluate ROPecker, as we find it is sufficient to identify useful sequences.

Binary	Setup	Flush	Syscall
diff	8	3	2
grops	4	3	4
lsof	12	2	3*
ltrace	4	2	2
grub-mkimage	4	4	3
strace	17	4	2*
pic	11	2	3
apt-get	14	3	2*
info	13	3	3*
apt-ftparchive	4	3	2

Table 1: The number of gadgets for the three steps in our kBouncer attack for binaries from `/usr/bin/`. Entries marked with an asterisk have success probability of $\geq 99.99\%$, the rest with 100%.

Our tool first enumerates all potential call-preceded gadgets. We implemented a simple symbolic execution framework to determine the effects of each of these potential gadgets. This system is not complete, but it models some of the effects of many common instructions.[5] It computes and outputs the path constraints that must hold to follow the conditional branches in a gadget. It also outputs the list of modified memory locations, accessed memory locations, and the new values of updated registers at the end of execution.

The tool returns a list of gadgets sorted by ease of use: gadgets with fewer conditional branches and fewer memory locations which must be valid rise to the top. Each gadget is marked with a hint on how it might be useful (e.g., that the gadget is a memory-load gadget, or that it computes the sum of two registers). It also provides us with a list of termination gadgets, sorted by ease of use and the number of other registers they clobber.

8.2 Fully Call-Preceded Attacks

How practical are fully call-preceded ROP attacks? Our measurements indicate that they are quite practical. The Q ROP compiler [26] is able to mount a ROP attack in 80% of binaries over 20KB in size. Given that only 6% of gadgets Q finds are call-preceded, we would expect that with 333KB of binary, we could achieve similar results. We actually found that it is possible to exploit 10 out of 10 programs we analyzed of size 70KB or larger.

We analyzed 10 binaries from `/usr/bin` on Ubuntu 12.04. In particular, we selected the first 10 binaries that have ASLR disabled and have more than 20k instructions (70KB binary size). In all 10 cases, we were able to find enough gadgets to mount a fully call-preceded history hiding ROP attack on kBouncer. Table 1 shows, for each of these ten binaries, the number of gadgets used for in each of the three phases of our ROP attack. Attacks marked with an asterisk have a success probability of $\geq 99.99\%$ due to the possibility of a module crossing a 32-bit boundary. All other attacks have a 100% success probability.

In each of these binaries, we use only the code present in the actual binary, not any other linked libraries. We are not arguing that these binaries are vulnerable to attack; we are only attempting to determine how much program text is required to mount fully call-preceded attacks.

We believe there to be two main reasons why we were so successful. First, we manually analyzed these binaries in order to construct a ROP attack, whereas Q is an automated tool. However, given Q's sophisticated analysis, we do not believe this accounts for all of the difference. We suspect that even though only 6% of gadgets are call-preceded, they have more diversity and thus are disproportionately likely to cover the space of different kinds of gadgets that are needed.

8.3 Modifying Real-World Exploits

We now evaluate the difficulty of modifying real-world exploits to bypass both kBouncer and ROPecker. To choose our exploits, we pick the ROP attacks that were shown to be prevented by kBouncer and ROPecker.

For kBouncer, we show how all four of these attacks can be modified so kBouncer will not detect them.

We finally modify the one real-world exploit which ROPecker is shown to prevent to bypass ROPecker.

8.3.1 kBouncer Exploits

We modified four real-world exploits to bypass kBouncer. None of the modifications to these exploits took us significant effort. Once we were able to reproduce the exploit on our machine, each exploit took under half of a day's worth of work to make it bypass kBouncer. Given the long and difficult exploitation development process, we do not think this is meaningfully harder, especially for well-trained exploit developers.

MPlayer Lite r33063. This program [19] had a stack-based buffer overflow vulnerability, which was exploited by overwriting the SEH pointer [20]. The `avcodec-52.dll` does not have ASLR enabled. This `dll` is 10MB, and contains plenty of gadgets: there were 748 potential termination gadgets with two or fewer conditional branches. The first of these that we tried worked, and was given previously in Fig. 3(b).

[5] The most important deficiencies in our tool are as follows: we implement only the thirty most-used instructions (covering 99% of instructions used in our binaries), we ignore segment registers, we do not track several of the flags set by instructions, and we do not properly handle referencing variable register widths. Despite this, we have found our tool to be accurate in the vast majority of cases.

Adobe Reader 9.3.4. This Adobe Reader exploit uses a sophisticated JavaScript vulnerability and was built on the Metasploit framework [1]. This exploit relied on `icucnv36.dll` having ASLR disabled. This `dll` is 10MB and has 130 available termination gadgets with two or fewer conditional branches. We created a ROP chain to call `VirtualProtect` on a page and verified that code on this page in memory could be executed.

Adobe Flash 11.3.300. An integer overflow caused this vulnerability in Adobe Flash. This exploit was also built with the Metasploit framework [2]. The exploit relied on `msvcr71.dll` having ASLR disabled. This `dll` is 300KB and has 64 available termination gadgets. In this exploit, we were able to successfully change a page to be executable and spawn another process.

Internet Explorer 8. The final exploit we modified was in IE8 and also used Metasploit [3]. This exploit was the most difficult for us to modify, and required a manual stack-pivot to a controlled location so that we could invoke `VirtualProtect` in a call-preceded manner. We relied again on `msvcr71.dll` to spawn another process.

8.3.2 ROPecker Exploits

ROPecker was built as a Linux kernel module and was shown to stop two exploits. One of these two exploits by the authors is to exploit a 20-line example C program with a trivial stack overflow from ROPEME [17]. The other exploit is a real-world exploit in *hteditor*, which has a published vulnerability [33] they verified they defend against. Because they only evaluate their defense on one binary, we have only this one binary to demonstrate our attack on. We evaluate our two methods of attack (repeated history hiding and evasion attacks) on this binary.

The public vulnerability disclosure included an exploitable version of the *hteditor* source. We downloaded this and compiled it for our system with stack protection disabled, as we want to test how well ROPecker defends against attack, not how well stack canaries work.

Evasion attack. We successfully mounted an evasion attack on hteditor. Our exploit required 12 gadgets. We split the attack into two 10-gadget segments, with the second segment calling `execv` by overwriting the GOT entry for `strlen` and finding a call-preceded intended call to it.

Repeated history hiding attack. We successfully mounted a repeated history hiding attack on hteditor assuming four pages in the executable set. Our attack consisted of three phases. In the first two phases we computed the address of `execv`, and in the third we called it. In the first phase, we were able to use a gadget twice that we loaded once.

9 Related Work

Randomization-based approaches. Address Space Layout Randomization (ASLR) and Address Obfuscation [5] were first introduced to make it more difficult to inject shellcode, and were later applied to the text segment to prevent ROP attacks. Shacham et al. demonstrated a de-randomization attack [28] on PaX ASLR.

Address Space Layout Permutation (ASLP) [16] is similar in many ways to ASLR but provides higher entropy by permuting the locations of functions. Other defenses extends this further by randomizing the addresses of individual instructions [15, 31]. Another technique replaces short sequences of instructions with alternate, functionally-identical, equal-length sequence, hindering an attacker's ability to use unintended gadgets [22]. A recent just-in-time code reuse attack [29] compiles ROP on the fly to bypass ASLR.

Control-Flow Integrity (CFI). Abadi et al. introduced control-flow integrity (CFI) [4] as a method of preventing attacks by restricting jump, call, and return instructions to follow the statically-determined control-flow graph of the program. Due to the difficulty of obtaining a precise control-flow graph of the program, many defenses choose instead to enforce a less precise policy. Often, this policy simply requires that returns be call-preceded, and indirect calls point to the beginning of functions [34, 6, 32].

The attacks presented in this paper show these CFI based defenses are weaker than previously thought. Since call-preceded ROP is possible, most of these defenses can be broken with that technique alone. Concurrent to this work, a detailed examination of attacks on many CFI-based schemes came to this same conclusion [14].

Runtime defenses. There are many other types of defenses that can best be described as runtime defenses. DROP [9] monitors the runtime behavior of the process and, nearly identically to ROPecker, if there is a long consecutive sequence of returns, each of which contain fewer than a fixed length, the program is killed. Our work in this paper constitutes a total break of DROP. ROPGuard [13] contains several heuristics to detect ROP attacks. One of these is the call-preceded defense introduced earlier. ROPdefender [12] implements a shadow-stack and verifies that all returns exist somewhere on the shadow-stack. Our work does not apply to shadow-stack defenses.

Recompilation-based defenses. Other defenses rely on recompilation to remove gadget from the compiled binary. G-Free [21] does this by removing unintended return instructions and encrypting return addresses, so that `ret`-gadgets become nearly impossible to use. The return-less kernel [18] entirely removes the `c3` byte (the opcode of `ret`) from all instructions, and replaces valid returns with a lookup into a table containing the valid return sites.

10 Conclusion

In this paper, we have presented three building blocks for ROP attacks that allow us to break two state-of-the-art ROP defenses. We demonstrate the practicality of our attacks by modifying real-world exploits to bypass these defenses.

More broadly, our work disproves two pieces of conventional wisdom: that ROP attacks only consist of short gadgets, and that ROP attacks cannot be effectively mounted in call-preceded manner.

Future defenses must take care to guard against attacks similar to ours. Specifically, we suggest two particular requirements for future defenses. First, defenses should argue either that they can inspect all relevant past history or, if they have a limited history, that their limited view of history cannot be effectively cleared out by an attacker. Second, defenses that defend against one specific aspect of ROP must argue that is a *necessary* component of one.

We believe an important open research question is to determine what properties are truly *fundamental* about ROP attacks that are different than typical program execution. We hope future work will explore how these fundamental differences can be exploited to create general-purpose defenses.

Acknowledgments

We gratefully acknowledge Matthias Payer, Michael McCoyd, Thurston Dang, and the anonymous reviewers for their helpful feedback. This research was supported by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-12-1-0040 and MURI award FA9550-09-1-0539, and by the National Science Foundation under grant CCF-0424422.

References

[1] Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow. `http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_cooltype_sing`.

[2] Adobe Flash Player 11.3 Kern Table Parsing Integer Overflow. `http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_flash_otf_font`.

[3] Microsoft Internet Explorer CButton Object Use-After-Free Vulnerability. `https://www.rapid7.com/db/modules/exploit/windows/browser/ie_cbutton_uaf`.

[4] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[5] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120, 2003.

[6] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[9] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.

[10] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 12–12, 2005.

[11] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against rop attacks. NDSS14, 2014.

[12] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[13] Ivan Fratric and Elias Bachaalany. ROPGuard. `http://code.google.com/p/ropguard/`.

[14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[15] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585. IEEE, 2012.

[16] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[17] Long Le. Payload already inside: Data re-use for ROP exploits. *Black Hat USA*, 2010.

[18] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208. ACM, 2010.

[19] Nate M. MPlayer (r33064 Lite) Buffer Overflow + ROP exploit. `http://www.exploit-db.com/exploits/17124/`.

[20] Brian Mariani. Structured exception handler exploitation. `http://www.exploit-db.com/wp-content/themes/exploit/docs/17505.pdf`.

[21] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[22] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[23] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[24] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Security & Privacy, IEEE*, 2(4):20–27, 2004.

[25] Marco Prandini and Marco Ramilli. Return-oriented programming. *Security & Privacy, IEEE*, 10(6):84–87, 2012.

[26] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[27] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[28] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[29] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[30] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[31] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[32] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[33] ZadYree. HT Editor 2.0.20 Buffer Overflow (ROP PoC). `http://www.exploit-db.com/exploits/22683/`.

[34] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.