2014 IEEE Symposium on Security and Privacy

Framing Signals—A Return to Portable Shellcode

Erik Bosman
Vrije Universiteit
Amsterdam
erik@minemu.org

Herbert Bos
Vrije Universiteit
Amsterdam
herbertb@cs.vu.nl

***Abstract*—Signal handling has been an integral part of UNIX systems since the earliest implementation in the 1970s. Nowadays, we find signals in all common flavors of UNIX systems, including BSD, Linux, Solaris, Android, and Mac OS. While each flavor handles signals in slightly different ways, the implementations are very similar. In this paper, we show that signal handling can be used as an attack method in exploits and backdoors. The problem has been a part of UNIX from the beginning, and now that advanced security measures like ASLR, DEP and stack cookies have made simple exploitation much harder, our technique is among the lowest hanging fruit available to an attacker.**

Specifically, we describe Sigreturn Oriented Programming (SROP), a novel technique for exploits and backdoors in UNIX-like systems. Like return-oriented programming (ROP), sigreturn oriented programming constructs what is known as a 'weird machine' that can be programmed by attackers to change the behavior of a process. To program the machine, attackers set up fake signal frames and initiate returns from signals that the kernel never really delivered. This is possible, because UNIX stores signal frames on the process' stack.

Sigreturn oriented programming is interesting for attackers, OS developers and academics. For attackers, the technique is very versatile, with pre-conditions that are different from those of existing exploitation techniques like ROP. Moreover, unlike ROP, sigreturn oriented programming programs are portable. For OS developers, the technique presents a problem that has been present in one of the two main operating system families from its inception, while the fixes (which we also present) are non-trivial. From a more academic viewpoint, it is also interesting because we show that sigreturn oriented programming is Turing complete.

We demonstrate the usefulness of the technique in three applications. First, we describe the exploitation of a vulnerable web server on different Linux distributions. Second, we build a very stealthy proof-of-concept backdoor. Third, we use SROP to bypass Apple's code signing and security vetting process by building an app that can execute arbitrary system calls. Finally, we discuss mitigation techniques.

I. Introduction

Signal handling has been an integral part of UNIX (and UNIX-like) systems ever since the very first implementation by Dennis Ritchie in the early 1970s. Signals are an extremely powerful mechanism to deliver asynchronous notifications directly to a process or thread. They are used to kill processes, to tell them that timers have expired, or to notify them about exceptional behavior. The UNIX design has spawned a plethora of UNIX-like "children" of which GNU Linux, several flavours of BSD, Android, iOS/Mac OS X, and Solaris are perhaps the best known ones in active use today. While each flavor handles signals in slightly different ways, the different implementations are all very similar.

We show that the implementation can be used as an attack method in exploits and backdoors, much like return oriented programming (ROP [24])—although the technique is different. The problem has existed, to the best of our knowledge undiscovered, for 40 years already. Moreover, now that advanced security measures like ASLR, DEP and stack cookies are making simple exploitation much harder, it probably ranks among the lowest hanging fruit available to an attacker.

In the tradition of 'weird machines' [7], we describe a technique for executing attacker-provided code in otherwise benign binaries. However, rather than executing shellcode directly, returning into the C library, or piecing together a program using ROP, we construct our weird machine by means of fake returns from signals. The technique, which we refer to as *sigreturn oriented programming*, is generic and we can use it both for exploits, backdoors, and system call proxies. Moreover, we prove that sigreturn oriented programming is Turing complete.

The key idea behind sigreturn oriented programming is that an attacker can abuse the way in which most UNIX systems return from a signal handler.

When the kernel delivers a signal, it suspends the process' normal execution and changes the user space CPU context such that the appropriate signal handler is called with the right arguments. When this signal handler returns, the original user space CPU context is restored. Specifically, a program returns from the handler using `sigreturn`, a 'hidden system call' on most UNIX-like systems, that reads a signal frame (`struct sigframe`) from the stack, put there by the kernel upon signal delivery. The frame contains all information needed for a safe return: the values of the registers, stack pointer, flags, etc.

The problem is that anyone who controls the stack is able to set up such a signal frame. By calling `sigreturn`, attackers may determine the next state for the program, and, as we shall see, chain together `sigreturn` and other system calls and to execute arbitrary code.

Like return-oriented programming (ROP), sigreturn ori-

DOI 10.1109/SP.2014.23

ented programming (SROP) is a generic technique and we will show how we used it in exploits, backdoors and system call proxies, and across a wide variety of operating systems. Compared to ROP, it has different preconditions that in certain cases are simpler to satisfy. For instance, SROP needs only a single gadget for the exploit and for several Linux, Android, and BSD distributions that gadget is always present, and better still, always located at a fixed location. In that case, attackers need not even know in advance the exact version of the executable and all the libraries, which greatly simplifies the attack when detailed reconnaissance is not possible.

For attackers, what is especially attractive about SROP compared to ROP, is its re-usability. Unlike ROP code, SROP programs are not very dependent on the content of the executable. Like traditional shellcode, this makes it possible to reuse the same SROP code across a host of applications. Moreover, the technique works on widely different instruction set architectures and operating systems. For example, we have implemented SROP successfully on 32 bit and 64 bit x86, as well as on ARM processors. Likewise, we successfully tested the technique on Linux (different distributions), Android, FreeBSD, OpenBSD, and Apple's Mac OS X/iOS. The program that we use to demonstrate Turing completeness of sigreturn oriented programming works on 32 bit and 64 bit Linux.

Contributions: In this paper we will describe a new type of weird machine and explain how to program it using sigreturn oriented programming. Specifically, we introduce:

1) a new generic exploitation technique, known as sigreturn oriented programming (SROP), that in some cases requires no prior knowledge about the victim application and yields reusable 'shellcode';
2) a novel and stealthy backdoor technique based on SROP;
3) a system call proxy to bypass Apple's iOS security model;
4) a proof that SROP is Turing complete;
5) possible mitigation techniques.

Applications: We demonstrate the practicality of our exploitation technique using a vulnerability found recently in the Asterisk web server. We show that using SROP, we can construct a single exploit which works for different versions of the same program on different distributions of Linux, including Debian Wheezy (released in May 2013), Ubuntu LTS 12.04 (the latest Long Term Support version of Ubuntu, released in 2012 and supported for five years), and Centos 6.3 (released in 2012 and supported for 10 years).

We then demonstrate the wider practical applicability of the technique by means of a stealthy backdoor. The backdoor is hard to find even with state of the art forensics tool. For example, even a complete memory dump of the process will not reveal any backdoor instructions, as all logic is hidden in the data in the form of an SROP program. Finally, we show how we can bypass Apple's well-known security model that consists of elaborate code vetting and prevents malicious apps from making it into the App Store. In our case, there is no need to add any malicious instructions to the application whatsoever, so it will be extremely hard to detect. However, provided with the right inputs, it will function as a system call proxy that can be programmed using sigreturn oriented programming.

Outline: The remainder of this paper is organized as follows. In Section II, we place SROP in the context of related work. In Section III, we discuss weird machines and how to program them, as well as the role of return-oriented programming. Signal handling is the topic of Section IV. Section V discusses the SROP technique in detail. We discuss our exploitation technique in Section VI, a stealthy backdoor in Section VII, and our iOS system call proxy in Section VIII. Turing completeness will be discussed in Section IX. In Section X we discuss possible mitigation techniques. Finally, we conclude in Section XI.

II. Related work

Our work fits in the general category of weird machines. The term weird machine was originally coined by Sergey Bratus and was quickly picked up by other researchers [7], [30]. It is a generic term to describe systems in which unwanted or unexpected powerful computations are found to be possible. For instance, researchers have shown that that it is possible to use weird machines constructed from arcane parts, such as ELF symbol relocation logic in the dynamic loader [26] and even the x86 virtual memory subsystem [9].

For this paper, Return-Oriented Programming (ROP) is most relevant [24]. ROP is an exploitation technique that allows attackers to execute code in the presence of security measures like non-executable stacks and code signing. As a precondition to a successful ROP exploit, the attacker needs to gain control over the stack and obtain valid code pointers. The attacker then manipulates the return address to jump to a sequence of instructions that ends with a return. As the attacker controls the stack, she can keep jumping to code gadgets and thus string together a program.

A variant of the same technique, known as jump oriented programming, uses jumps instead of returns [3]. Finding the appropriate gadgets and stringing them together is hard, but researchers have shown that it is possible to construct ROP compilers [23], [13] to make this easier. Similarly, modern protection mechanisms like address space randomization [20], [1], [25] make it hard to find the addresses of the code snippets. On the other hand, given an initial code pointer, it is often possible to extract the remaining code pointers necessary for a successful exploit [27]. Also, in Linux and Windows, executables with fixed load addresses, and libraries incompatible with ASLR, may lead to parts of the address space that are constant [22].

Since ROP and JOP have quickly become popular with attackers, many research groups have tried to mitigate the issues, for instance by trying to remove useful gadgets [11], [17], permuting the order of functions [2], [10], or dynamic binary instrumentation [6], [5]. Also, researchers have proposed to use in-place code randomization with low overhead [19] and by monitoring branch histories [18]. Even though none of these solutions have stopped attackers from using ROP, they keep raising the bar. Many of the mitigations do not work for SROP. For instance, the gadgets used by SROP are essential for the functioning of the binary and cannot be removed. We believe that SROP may be among the most convenient methods of attack even against programs that apply all common security measures.

At a superficial level, sigreturn oriented programming shares some of the characteristics of ROP. We will see that sigreturn oriented programming also requires setting up the appropriate values on the stack and 'returning' to a value determined by the attacker. But the technique is different. Specifically, SROP uses fake signal frames and does not really depend on finding gadgets and stringing them together. For this reason also, SROP code has better re-usability.

The use of signals themselves in exploitation has been thoroughly documented by Michael Zalewski in: "Delivering signals for fun and profit" [32]. The author shows that there are many pitfalls to consider when programming signal handlers. The pre-emptive nature of signals can lead to the use of data structures while they are in an inconsistent state, corrupting them in the process, or to make an application do unexpected things while in a state with elevated permissions. Our work is different, in that we use fake signal frames as partial instructions in a weird machine.

A user space call which is somewhat similar to the `sigreturn` system call is `longjmp`. In "Bypassing stackguard and stackshield" [4], overwriting a pointer to a user context subsequently used by `longjmp` is mentioned as an exploitation vector. It is noted however, that the existence of such a pointer in an exploitation context is extremely rare.

Microsoft Windows does not implement POSIX signals. Its fault handling mechanism is designed to integrate well with C++'s exception handling. Through a mechanism called Structured Exception Handling (SEH), it allows functions to unwind the stack and do some processing until the exception is caught in an earlier stack frame. As exception handler pointers are put on the stack, this has been a steadfast overwrite target for stack buffer overflows on Windows [12] and several mitigation techniques have been proposed and implemented [15]. However, since SEH on Windows does not return to the state before the handler was executed, they cannot provide a mechanism for SROP-like exploits.

An independent effort to attack iOS devices that also builds on benign apps becoming evil is provided by Jekyll [31]. Like our work, the authors deliberately introduce vulnerabilities in the app, so that they can easily change the control flow later, by exploiting it by means of a ROP exploit. Likewise, we show that it is now *easy* to circumvent all of iOS's defensive mechanisms, including DEP [14], ALSR [20], [1], sandboxing, app review and code signing. Unlike Jekyll which needs to carry its own gadgets, the gadget for sigreturn oriented programming is already present. Thus, even detection techniques that specifically scan for Jekyll-like functionality are no longer effective. Finally, we can use sigreturn oriented programming in a post-exploitation scenario. For instance, we can use the system call proxy for a jailbreak—allowing attackers to jailbreak via a root process that was not written by them.

In the next section, we will revisit the current trend among attackers to move away from regular shellcode injection toward attacks based on code reuse. We will argue that because of this, exploitation is getting harder and that SROP may be a useful new weapon in the hands of an attacker—removing some of the obstacles that an attacker encounters in modern systems.

III. On the construction of weird machines (or: why exploitation is getting harder)

Sigreturn oriented programming is not unlike other exploitation techniques where attackers execute foreign 'code' in an existing program. Not so long ago, doing so was relatively straightforward. Typically, attackers managed to find a buffer overflow vulnerability on the stack which allowed them to overwrite the return address. All they needed to do was make the return address point to their own machine code which would be stored in a buffer or environment variable. As soon as the function returned, the program would continue executing the attackers' code.

1) Code reuse attacks (ROP, JOP, and ret-into-libc): With modern protection measures, such traditional exploitation of memory corruption bugs, where machine code is injected by the attacker and directly executed, has become rare. Specifically, data execution prevention (DEP [14]) features, present in practically all modern CPUs and operating systems, separate machine code and writable data. In other words, attackers can no longer execute their shellcode directly. To bypass such security measures, they must resort to different exploitation techniques. Rather than injecting regular machine code, modern attacks typically re-use logic or code from the executable itself using ret-to-libc [28] or return oriented programming [24]. In this way, attackers can construct strange automata to do their bidding.

Constructing such an automaton is not easy. Over the past few years, the security research community has especially focused on generic techniques like *return oriented programming* (ROP) [24] and *jump oriented programming* (JOP) [3]. These techniques make use of chunks of executable code present in the program itself. Chained by indirect control flow instructions, these chunks thread together a sequence of low level instructions which do what the attacker wants.

An attacker that controls an application's stack because of a buffer overflow may look in the original application's binary for small sequences of instructions that do something interesting and end with a return instruction. For instance, a sequence to add two registers, or to load or store a register, etc. These sequences are known as 'gadgets'. If the attacker manages to divert the control flow to one of the gadgets, the gadget will execute the first small part of the attacker's code and then execute a return. Since the attacker controls the stack, the address to return to is also under the attacker's control. So the attacker returns to another gadget. By chaining together gadgets, the attacker can execute arbitrary code, written for an instruction set that consists of the gadgets and a stack pointer that functions as a strange sort of program counter.

While such a procedure may sound straightforward, a working ROP exploit is often highly complex. For instance, a ROP attack may require an attacker to manipulate the state of the program to prepare it for exploitation, by (1) making it leak information about code pointers (for instance, by means of a format string attack), (2) lining up heap objects in a specific way to pave the way for a dangling pointer exploit (using advanced heap feng shui [29]), (3) gathering enough useful code gadgets from the binary to chain together the attacker's final program (which typically assumes that the exact versions of the executable and libraries are known), and (4) positioning the appropriate data on the stack, and (5) diverting control to the first gadget.

The ability of a program to go beyond its specification allows it to act as a *weird machine* [7], [30]. This machine can thereafter be manipulated, programmed by an attacker just like any other programmable machine. The attacker's input now serves as its (rather peculiar) machine code, giving her control over the program's execution, far beyond its intended use.

What the attacker wants: Some techniques for programming weird machines are very application specific. They depend very much on the flaw(s) that make up a vulnerability in a single application and cannot be re-used for the next bug. Others, like heap feng-shui with Javascript [29], apply to a whole class of applications and considerable effort is spent on making them applicable across systems. Clearly, reusable techniques are more valuable than a trick that can be used only once.

From an attacker's perspective, we therefore distill the following objectives for the construction and use of weird machines:

1) **Re-usability**: reusable techniques are better than one-offs. Ideally, we would like to be able to reuse the same exploits for multiple programs.
2) **Simplicity**: the less manipulation is needed *a priori*, the better. If the list of manipulations is long and complex, the attack may be difficult to pull off for some programs.
3) **Generality**: a technique that can be used for exploits is good, but it is even better if it can also be used for backdoors and other purposes.
4) **Variety**: the more choice for exploiting a program, the better. If we have more than one attack vector for a program with different preconditions, the probability of finding one that fits a target program increases.

As explained above, a typical exploit in reality consists of a sequence of different techniques, where more application-specific code gymnastics bootstrap more generic exploitation methods such as ROP, and then possibly the attacker's native code. Again, the more generic the technique, the better the exploit code can be re-used for other vulnerabilities.

2) ROP makes attacks harder: As we saw earlier, exploitation techniques have evolved in answer to the protection measures that are now employed on modern systems: direct shellcode execution has become rare, and ROP exploits have become popular. Unfortunately for attackers, ROP is not such an easy drop-in solution as normal shellcode used to be, because useful code-snippets and their addresses are different for every new binary for which an exploit is being written.

It is true that ROP compilers make it possible to automatically generate a useful ROP chain for most target binaries, but even with state-of-the-art ROP compilers, return-oriented programming complicates the attacker's life significantly. With a traditional stack overflow and without data execution prevention, the same exploit could very easily work across multiple binaries with the same vulnerability. Now, however, the attacker needs to know exactly which binary the victim uses to feed it to a ROP compiler, or possibly even build a ROP chain manually. It may be difficult to determine the exact version of a binary because an application has had several updates which may or may not have been installed. In the worst case, the binary is a custom build with an unknown compiler and unknown compile flags. Even if the attacker can determine the version, it still is a lot of work to do this for every binary.

The other pre-conditions for a successful ROP attack are important as well. Besides control over the stack and an initial control-flow diversion, the executable needs to leak the address of a code pointer which, depending on the application, may be hard.

In the remainder of this paper, we will see that sigreturn oriented programming scores well on all of the above four criteria. SROP code is portable [objective 1], the attack is simple (e.g., it requires a minimal number of gadgets that in quite a few systems can be found at fixed locations) [objective 2], it has many applications [objective 3], and it enlarges the attackers' repertoire with different preconditions [objective 4].

IV. Signal delivery on UNIX systems

Signals have been an integral part of UNIX systems almost since its inception [21]. Initially little more than a convenient abstraction around hardware interrupts, the mechanism was later adapted as a generic system, able to receive notifications both from the kernel and from other processes. By registering a signal handler function, a process can deal with asynchronous notifications outside of its normal control flow.

Sigreturn oriented programming requires a thorough understanding of how signals are handled on UNIX systems. In particular, we will zoom in on the way the system restores the process state upon returning from a signal handler.

A. Delivering the signal

When a kernel delivers a signal to a process, the normal execution flow of the process is temporarily suspended. Specifically, the kernel changes the user space CPU context such that a previously registered signal handler function is called with the right arguments. When this signal handler returns, the original user space CPU context is restored.

In true UNIX fashion, this is implemented in an elegant way that requires no bookkeeping on the kernel side. The suspended user context is simply saved on the process' stack and restored from the stack when the signal handler returns. Initially, this was done simply in the interrupt trap handler (Fig. 1). But the introduction of virtual memory and hardware memory protection made this impossible, as a user-space signal handler could no longer directly return to the interrupt routine. This led to the introduction of the `sigreturn` system call in 4.3BSD. `Sigreturn` takes as first argument a pointer to the user context to be restored. A piece of trampoline code is placed in the user address space which first calls the signal handler and then does a `sigreturn` system call to signal the kernel it should restore the old user space context.

Linux does something similar to the BSDs, except that it executes the signal handler directly. When the signal handler returns, it returns to an address written there by the kernel. A stub at this address is then responsible for calling `sigreturn`. Unlike on BSD (and iOS/Mac OS X), there is no argument to `sigreturn`. The kernel simply loads the user context from the stack.

Figure 2 shows the top of the stack of a 64 bit x86 Linux process when a signal handler exits.

B. Sigreturn and Data Execution Prevention

The `sigreturn` calling trampoline has to be in user space memory. It used to be custom for kernels to write the trampoline code together with the user context on the stack. A returning signal handler would just jump to this code on the stack. However, this requires an executable stack, allowing for easy shellcode injection on the stack. Nowadays, depending on the architecture, the `sigreturn` trampoline code is either provided in an executable memory page provided by the kernel, allowing the kernel to know its location, or it resides somewhere in libc.

```
NSIG = 0

# interrupt vector
tvect:
    mov r0,-(sp);
    # load signal handler function
    mov dvect+[NSIG*2],r0;
    br 1f;
    NSIG=NSIG+1
    # repeated a total of 20 times (for 20 signals)
        ...
1:
    # push the register state on the stack
    mov r1,-(sp)
    mov r2,-(sp)
    mov r3,-(sp)
    mov r4,-(sp)
    # call the signal handler
    jsr pc,(r0)
    # restore registers
    mov (sp)+,r4
    mov (sp)+,r3
    mov (sp)+,r2
    mov (sp)+,r1
    mov (sp)+,r0
    # return from interrupt
    rtt
```

Figure 1. Excerpt from s5/signal.s, UNIX V6 interrupt routine (comments added for clarity). We see that even in these early versions, the suspended CPU state was stored on the stack.

V. Sigreturn Oriented Programming

In this section, we discuss the key ideas behind sigreturn oriented programming. To do so, we first introduce the way Linux and other UNIX flavors return from signals in some detail and then discuss how `sigreturn` is vulnerable to abuse.

A. Signal delivery on Linux

At the new top of the stack, the kernel writes the code address that will become the return address for the signal handler. This code address points to a small stub which does nothing more than invoke the `sigreturn(0)` system call. This `sigreturn()` call undoes everything that was done in order to invoke the signal handler (changing the process's signal mask, switching stacks). Thus, it restores the process's signal mask, switches stacks, and restores the process's context (registers, processor flags)—making the process resume execution exactly at the point where it was interrupted by the signal.

B. The sigreturn system call on Linux

While sigreturn is a system call, called like any other, it is special in the sense that no user space program ever needs to be aware of its ABI. The kernel is responsible for setting

up the user space stack in such a way that it is eventually called.

As mentioned earlier, the trampoline code invoking sigreturn used to be on the stack in the *signal frame* in the `pretcode` field, but because executing this trampoline requires an executable stack, it is no longer used on recent kernels. Interestingly, vestiges still exist on Linux i386 to help gdb identify the signal frame. In reality, however, the stub has since moved to the *virtual dynamic shared object (vdso)*, a kernel-supplied piece of code mapped in every process' address space. On x86-64 Linux, the stub is present in libc itself. It must be supplied to the kernel when registering signals using the `sigaction.se_restorer` field. In both cases, the `sigreturn` stub address is normally randomized by ASLR [20], [25].

When `sigreturn(0)` is called, the kernel will use the user space stack pointer to find the *signal frame* which it had stored there previously and load the original user context into the CPU's registers. In this way, it restores the original context for the interrupted process.

Figure 2 shows a signal frame that a 64 bit Linux kernel would place on the stack. We see that the frame contains register values, including the stack pointer (RSP), the instruction pointer (RIP), the segment registers (CS, FS, and GS) and the flags. The return address is at the top of the frame, followed by the user context, and the registers. The regular registers may hold arguments and we will use them in this way also.

The floating point state pointer points to the saved floating point unit state. It is only important if there is any floating point state to restore, i.e., if there were any floating point operations before the signal arrived. If the pointer is NULL, on the other hand, the kernel assumes that there were no such operations and ignore it.

Signal frames on other architectures look very similar, albeit that the register context stores by definition other, architecture-specific registers and values.

C. Sigreturn on other UNIX flavors

Different UNIX and UNIX-like systems implement returning from signals in slightly different ways. On BSD, Mac OSX and iOS, `sigreturn` is similar, except that it *is* part of the ABI and that it takes the location of the `struct sigframe` as first argument. On OpenSolaris there is no `sigreturn`; restoring the original context happens in user space and is handled completely by libc, which provides a wrapper around signal handlers and restores the user space context. While other operating systems also provide interesting avenues for exploitation, we will restrict ourselves to various flavors of Linux, BSD, and iOS in the remainder of the paper, often using Linux as an illustrative example.

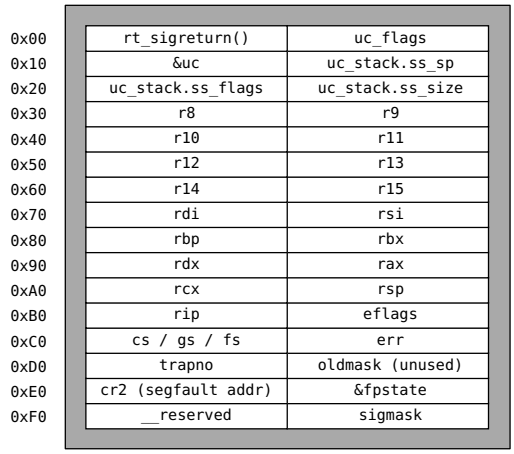

Figure 2. The signal frame as it looks like in Linux 86-64

D. Abusing sigreturn

Restoring the context of an interrupted process by loading a previously saved stack frame is convenient because it relinquishes the responsibility of the kernel to keep track of the signals it delivered. However, it also has a major drawback: the kernel does not keep track of the signals it delivered. In other words, there is no way of telling whether a `sigreturn` is legitimate.

By setting up a correct `struct sigframe`, loading the right system call number, and executing a system call instruction, an attacker can trivially fool the kernel into acting like a signal handler just finished. In that case, the kernel will load a user space context constructed by an attacker from the stack.

In the remainder of this paper we will explore the unintended consequences of being able to do a sigreturn on arbitrary data.

VI. SROP FOR EXPLOITATION

As mentioned, attackers can use sigreturn oriented programming for different purposes. First, we will outline two exploitation techniques, a simple generic technique followed by a more flexible method for 64 bit Linux processes. This second method is more complicated, but has weaker preconditions. In later sections, we discuss other uses of SROP.

A. A simple `execve()` SROP exploit

The final result of many UNIX exploits is often an attacker controlled shell. In this section we will outline an exploit which will call `execve` to start a shell with arbitrary arguments.

For any exploitation to be successful, certain preconditions have to be met. For instance, for direct shellcode execution, attackers must be able to load their code in executable memory and divert control to this buffer. ROP requires code pointers, gadgets, control of the stack, a control

flow diversion, etc. Likewise, successful SROP exploitation using this technique is possible if the attacker satisfies the following pre-conditions:

1) The attacker should have control over the instruction pointer (for example due to a return instruction on the overwritten stack).
2) The stack pointer should be located on attacker controlled data and $NULL$ bytes must be allowed (e.g., in an overflow). For BSD, Mac OS X and iOS, a function pointer overwrite with a user-controlled buffer as first argument is also a possibility. In this case there is no need for any user controlled data on the stack.
3) The attacker knows the address of a piece of data controlled by the attacker. This could be the overwritten stack, but does not necessarily have to be the same location.
4) The attacker knows the location of code calling `sigreturn`, or `syscall`, in case the attacker can control the CPU register which passes the system call number.

B. Finding a sigreturn gadget

As with return oriented programming, SROP needs some knowledge about the location of code in a process' address space. But unlike ROP, SROP really only needs to know the location of a single gadget, namely, the call to `sigreturn`. In our exploit we also make use of an extra gadget which does arbitrary system calls, but this additional gadget is contained in the `sigreturn` gadget. For this, we simply skip the instruction where the system call is loaded. In some cases we may be able to control the CPU register responsible for passing the system call. In that case we don't need a `sigreturn` gadget, and a `syscall` gadget will suffice as will be elaborated in our second exploitation technique.

As it turns out, it is surprisingly easy to find these gadgets on some architectures. On FreeBSD 9.2 on x86-64, there is a fixed memory page containing `sigreturn`. Linux on ARM, before version 3.11 (this includes all current versions of Android as well as the latest long term stable version (3.10) has a fixed `[vectors]` map with sigreturn gadgets. Furthermore, many versions of Linux on x86-64 have a fixed [vsyscall] map at with a `syscall & return` gadget (see Section VI-D).

If `sigreturn` is located in libc, and there are no known gadgets beforehand, an attacker might need to leak libc code addresses in order to obtain a sigreturn gadget. On systems which do library pre-linking, sigreturn gadgets may be the same system-wide, which allows local exploits to find the right gadgets. Table I shows the location of useful gadgets in various Operating Systems.

C. Executing system calls

As most architectures pass system call parameters to the kernel through registers (a notable exception being BSD on i386), doing a `sigreturn` allows us to do an arbitrary system call. By setting our stack frame's instruction pointer to the address of a `syscall` instruction and filling in the registers that pass the system call number and its arguments we effectively load a state in which a system call of our choice gets executed.

Operating system		Gadget	Memory map	
Linux	i386	`sigreturn`	[vdso]	
Linux $<$ 3.11	ARM	`sigreturn`	[vectors]	0xffff0000
Linux $<$ 3.3	x86-64	`syscall & return`	[vsyscall]	0xffffffffff600000
Linux $\geq$ 3.3	x86-64	`syscall & return`	Libc	
Linux	x86-64	`sigreturn`	Libc	
FreeBSD 9.2	x86-64	`sigreturn`		0x7fffffff000
OpenBSD 9.2	x86-64	`sigreturn`	sigcode page	
Mac OS X	x86-64	`sigreturn`	Libc	
iOS	ARM	`sigreturn`	Libsystem	
iOS	ARM	`syscall & return`	Libsystem	

Table I
SIGRETURN AND SYSCALL GADGETS AND THEIR LOCATION

Since we know the location of some data controlled by the attacker, we can now do an `execve` system call to, say `/system/bin/sh` on Android, using pointers to our data for `execve`'s filename and argv parameters and using the fixed `sigreturn` & `syscall` gadgets from the `[vectors]` page.

D. SROP exploitation on Linux x86-64 using a system call chain

Up until now we have assumed that we knew about memory locations with attacker controlled data and that we had knowledge of a `sigreturn` gadget. For the next exploitation method we won't have to know exact locations of attacker controlled data in the address space and we will not require any knowledge about code from the application itself. This exploitation technique is targeted at x86-64 systems running a Linux kernel older than version 3.3 and using this method, we will exploit different versions of a vulnerable Asterisk server using the exact same exploit on different systems.

Our new pre-conditions are:

- The stack pointer should be located on attacker controlled data and $NULL$ bytes must be allowed (e.g., in an overflow).
- The attacker should have some control over `RAX`. Specifically, `RAX` should contain the value 15.
- The attacker should have control over the instruction pointer `RIP` (for example due to a `RET` instruction on the overwritten stack).
- The attacker should know the location of a (any!) single writable page in memory. The location or content of the binary's code is not important. If the attacker manages to leak a writable address, the executable can even be position independent and the exact identity of the binary completely unknown.
- The target's system implements native `vsyscall`, which is the default on pre-3.3 kernels such as those

```
$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 fe:00 786514            /bin/cat
0060b000-0060c000 r--p 0000b000 fe:00 786514            /bin/cat
0060c000-0060d000 rw-p 0000c000 fe:00 786514            /bin/cat
01148000-01169000 rw-p 00000000 00:00 0                 [heap]
7ff9bbc54000-7ff9bbdd1000 r-xp 00000000 fe:00 659315    /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbdd1000-7ff9bbfd1000 ---p 0017d000 fe:00 659315    /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd1000-7ff9bbfd5000 r--p 0017d000 fe:00 659315    /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd5000-7ff9bbfd6000 rw-p 00181000 fe:00 659315    /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd6000-7ff9bbfdb000 rw-p 00000000 00:00 0
7ff9bbfdb000-7ff9bbffb000 r-xp 00000000 fe:00 659338    /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bc1ef000-7ff9bc1f2000 rw-p 00000000 00:00 0
7ff9bc1f8000-7ff9bc1fa000 rw-p 00000000 00:00 0
7ff9bc1fa000-7ff9bc1fb000 r--p 0001f000 fe:00 659338    /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bc1fb000-7ff9bc1fc000 rw-p 00020000 fe:00 659338    /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bc1fc000-7ff9bc1fd000 rw-p 00000000 00:00 0
7ffff7c2c000-7ffff7c4d000 rw-p 00000000 00:00 0         [stack]
7ffff7c61000-7ffff7c62000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Figure 3. Vsyscall is mapped in a process' address space

```
0xffffffffff600000:  mov    $0x60,%rax   ; gettimeofday()
0xffffffffff600007:  syscall
0xffffffffff600009:  retq
0xffffffffff60000a:  int3
...
0xffffffffff600400:  mov    $0xc9,%rax   ; time()
0xffffffffff600407:  syscall
0xffffffffff600409:  retq
0xffffffffff60040a:  int3
...
0xffffffffff600800:  mov    $0x135,%rax  ; getcpu()
0xffffffffff600807:  syscall
0xffffffffff600809:  retq
0xffffffffff60080a:  int3
...
```

Figure 4. The vsyscall page contains the `syscall & ret` gadget multiple times

used on Debian 7.0 (2013), Ubuntu Long Term Support (2012) and CentOS 6 (2012).

- The attacker controls data sent to a file or socket with a known file descriptor number.

In general terms, the SROP attack works by repeatedly setting up fake signal frames and returning from a "signal" to regular system calls and back. To explain how this is possible, we first describe the details of system call execution.

1) Useful gadgets on Linux x86-64: vsyscall: Vsyscall is a fast system call interface for 64 bit Linux. It was created to speed up certain time-sensitive system calls. Instead of using the syscall instruction, an application could simply jump to static addresses in a static page set up by the kernel (see Figure 3). The kernel provided user space code at these addresses which implemented `time()`, `gettimeofday()` and `getcpu()`. Any privileged data needed for these system calls to complete (such as the current time,) was provided by the kernel in data structures in the same page.

This turned out to be a security nightmare, as it provides attackers with a heap of useful gadgets for exploitation at fixed addresses, identical for every running process. These gadgets include `syscall & ret`: a syscall (`0f 05`) followed by a return (`c3`). For this paper, we are particularly interested in this `syscall & ret` gadget, because it allows one to execute a system call of choice, provided we can set the appropriate system call number in `RAX` (see Figure 4).

While gadgets seem to differ across *distributions*, useful gadgets appear to stay the same after kernel security *updates*. In other words, for systems running stock distribution kernels, it should be easy to determine the addresses of gadgets—even remotely! In addition, as the kernel keeps track of wall-clock time in the *same page* as the executable code, a patient attacker could just wait for the binary representation of wall-clock time to contain a (small) gadget of his/her choice and then jump to (the least significant bytes of) the wall-clock time field.

Due to its security risks, `vsyscall` was deprecated in Linux 3.1 and by default the fast user space routines now just call their normal system call counterparts. While this eliminated some harmful gadgets, due to its simplicity, it did result in the remaining gadgets, such as `syscall & ret` having *stable addresses across all distributions*. The reason is that the C file containing `vsyscall` was replaced with an assembly file that contains several `syscall & ret` gadgets. The assembly file always generates the same binary.

At the same time, the Linux developers added a second `vsyscall` emulation mode, which emulates vsyscalls using a trap-based mechanism. As a result, we cannot make use of the system call gadgets from this page. Note, however, that for many kernels this is not the default configuration[1]. Even if, there is no fixed location for system call gadgets, we may still find one in the executable itself—after all, we only need one small gadget. In that case, we lose the property of not needing reconnaissance, but the exploit still works. As a long-term release, the 3.2 kernel is present in many distributions today, such as Debian 7.0 (released in 2013, supported for 10 years) and Ubuntu 12.04 Long Term Support (released in 2012 and supported for 5 years)

E. Bootstrapping to arbitrary code execution

In this section, we show how the `sigreturn` can help the attacker to execute arbitrary code. To do so, we make use of a chain of system calls and sigreturns that serve as an example. There may well be other sequences that an attacker can use to achieve the same effect.

The first thing we do is create a fake signal frame and trigger a `sigreturn`. We first explain what that first signal frame looks like.

In order to execute a successful `sigreturn` system call, we must ensure that the kernel does not trip over any bad values in the signal frame. The first requirement is that the code segment register is restored correctly. On x86-64, when running in 64bit mode, the code segment register should contain the value `0x33`. The second requirement is that the fpstate pointer is not a wild pointer. Fpstate points to the saved floating point unit state and if its address not valid, or the saved floating point state is not correct, our application

[1]It is certainly not default for all kernels prior to Linux 3.3. and even post-3.3 kernels allow one to boot without emulated `vsyscall`.

will crash. This seems like a problem, since our assumption is that we do not yet know of any attacker controlled data on a known address. Luckily, when fpstate is $NULL$, Linux assumes no floating point operations had been used before the signal arrived. In this case, it clears the FPU state is and sigreturn succeeds.

When we execute `sigreturn`, we have complete control over the registers, as well as over the program counter (RIP.) But we have also lost any chance of using any ASLR information present in the registers as they have been overwritten. This is why we need the address of a writable page. We will point the stack pointer in our signal frame to the bottom of this page. By filling in the necessary registers and pointing the program counter to our `syscall & ret` gadget, we can set up and subsequently execute any system call we like (see also Figure 5).

We will use this power to set up a `read()` system call. The `read` will read in attacker data and store it under the stack pointer. When read finishes, the attacker's data will serve as the return address. In our case, the attacker points it *again* to the `syscall & ret` gadget.

Steps 1 and 2: `sigreturn` and `read`

Summarizing, in our first two steps, we create a fake signal frame on the attack, as explained above. In the signal frame the RSP value will point to the writable page, the value for RAX will be 0 (indicating a `read` system call), and there will be appropriate values in the registers that serve as the arguments to the `read` system call. Then, the attacker diverts the program's control flow to the `syscall & ret` gadget to execute the system call and read the attacker's data on the new stack (the writable page). Because the attacker controls the return address, he can simply return to the `syscall & ret` gadget—not unlike what one would do in regular ROP.

At this point the attacker has no choice but to keep all the system call arguments the same, as the kernel does not change them during the execution of a system call. But RAX will contain the number of bytes read. This is important and the attacker uses it to select the next system call to execute. As mentioned earlier, the RAX register indicates which system call to execute.

Step 3: a necessary NOP

Specifically, the attacker chooses to read 306 bytes. Not only does this give quite some data which is now at a known location, but 306 also happens to be the system call number for `syncfs(int fd)`—our third step in the exploit. The `syncfs()` system call takes a file descriptor as first argument and flushes all disks belonging to this descriptor. Since our file descriptor is a socket, this will effectively be a no-op returning 0 in RAX. The attacker again makes sure to return to the `syscall & ret` gadget.

Step 4: another read to set RAX

On x86-64, the value 0 happens to be the system call number for `read`, allowing the attacker to again read data onto the new stack—our fourth step. This time the attacker makes sure to send only 15 bytes, so that the value of RAX is now 15.

Steps 5 and 6: a `sigreturn` to execute anything we like

As mentioned earlier, 15 is also the system call number for `sigreturn`, so we are back to where we started, but with an important difference: there is data controlled by the attacker at a known address. The following `sigreturn`, our fifth step, is again free to load an arbitrary system call into the registers, which enables the attacker to do anything he wants. For instance, he can execute an `mprotect` system call and jump to traditional shellcode, or an `execve` with the right arguments to spawn a shell.

F. Exploiting the Asterisk web server

We have tested our exploitation technique on a recent version of Asterisk that is vulnerable due to an unbounded stack allocation bug (CVE-2012-5976). The vulnerability has been described in depth at EIP blog [8]. Our exploit is entirely new and targets multiple binaries.

The unbounded stack allocation vulnerability occurs when a pre-authentication HTTP POST request to Asterisk's web management console allocates HTTP post data on the stack. It uses the `content-length` header sent by the client to determine how much data should be allocated. Yet it does not check whether this size is within reasonable bounds. Specifying a `content-length` of about the size reserved for the stack allows us to 'jump' with our stack pointer to the stack of a second thread in the asterisk process. Having jumped to the second thread's stack, we can start sending our post data, which will promptly overwrite the second stack.

By making sure this second thread is also initiated by us, we can overwrite this thread's stack while it is waiting on a blocking read, When we are done corrupting the stack, we send 15 bytes to the thread with the corrupted stack and when it returns to our `syscall & ret` gadget, it `calls sigreturn`, setting in motion our bootstrap method from Section VI-E.

We still need to fulfill our two remaining requirements: we need to control a file descriptor and we need to know a writable page. For the writable page we guessed a page in the binary's data section. By default, with non position independent executables compiled with gcc, the data section comes directly after the code section, and the code section starts at a fixed offset of 0x40000. The size of the code section does not vary much between different versions of Asterisk compared to the absolute size of the data section, therefore it is quite easy to guess a writable page in the Asterisk binary.

In order to pick the right file descriptor, we open a large number of connections to the vulnerable section and send the same data over all sockets. By picking a high value for the file descriptor we can be fairly sure that we have selected a socket that we opened.

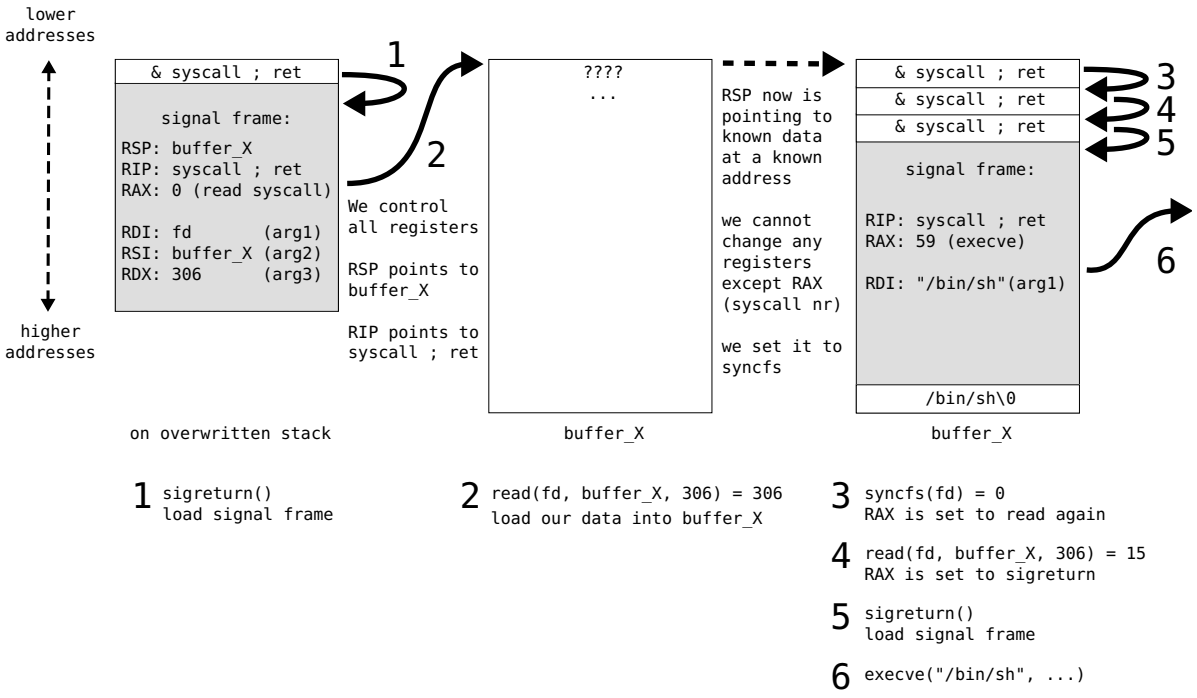

Figure 5. Steps involved in the Linux x86-64 SROP exploit

We tested this exploit on three different vulnerable versions of the Asterisk program on different Linux distributions: Debian Wheezy (released in May 2013), Ubuntu LTS 12.04 (the latest Long Term Support version of Ubuntu, released in 2012 and supported for five years), and Centos 6.3 (released in 2012 and supported for 10 years).

The exploit worked on all Linux distributions we tried. Moreover, the re-usability of the exploit is hinted on somewhat by the fact that the code for all three versions was almost exactly the same. The only difference, was that the `syscall & ret` gadget at Centos was at a slightly different location.

VII. SROP AS A BACKDOOR

Another possible use for sigreturn oriented programming is as a stealthy backdoor. By injecting signal frames into a process' address space and by either creating an extra thread in a process, or by simply replacing a process' execution with our own, it is possible to keep a presence on a system, while appearing to have left.

Developers of backdoors are keen to avoid detection. Unfortunately, injected shell code will look suspicious when a memory dump is viewed with forensics tools. Hiding all logic in data seems to be more stealthy. While in principle this can also be done using ROP, we will show that for sigreturn oriented programming it can be done in in a completely generic way, which works for all processes and requires no complex ROP compiler.

For our exploitation example, we assumed that the only gadget available to us was `syscall & ret` and this was provided to us as a non-ASLR gadget by the vsyscall page. For our backdoor we will no longer be depending on the vsyscall page. We will also no longer require the architecture to be 64 bit x86 as we need not use the x86-64 specific bootstrap method of going from one system call to another which made the return values of one system call the system call number of the next.

In our backdoor scenario the attacker uses `ptrace()` to inject a weird machine into a victim process. With `ptrace()` it is trivial to find a `syscall & ret` gadget as it is possible to trap on system calls. We will also assume that we have a complete `sigreturn()` gadget at our disposal. The gadget first loads the sigreturn system call number before it does a system call. This gadget can easily be found by sending the traced process a signal for which it has registered a handler, prompting the kernel to set up a signal frame with the `sigreturn()` gadget at the top of the stack.

Using our `syscall & ret` and `sigreturn()` gadgets and by faking some signal frames, we can create a chain of system calls. Each frame setting up the registers to do a certain system call while pointing the stack pointer at the next frame, each with a `sigreturn()` gadget at the top. Just like with ROP, it is in a sense the stack pointer which acts as an instruction pointer, only now the stack pointer always points at a complete user context state. While all we

do is execute a number of system calls (possibly in a loop) it is surprisingly simple to create complex behavior.

To demonstrate this, we have created a backdoor automaton that waits for a given file to be accessed. This file could for example be an obscure file on a publicly accessible web server. When this file is read, a listener TCP socket is created and if someone connects to this socket it spawns a shell connected to this socket. If no-one connects within 5 seconds, the listener stops listening until the trigger file is accessed again.

The assumption is that a remote party can easily cause the system to do a read operation on a file that otherwise is rarely read, for example a hidden file in a web document root. Only after this file is accessed, it will be possible to make a connection to the machine though a socket, something that otherwise would be very easy to spot.

To construct our backdoor, we chain together a string of signal frames executing system calls, relying on system call blocking semantics for our logic, as shown on the left-hand-side of Figure 6.

In particular, we make use of the `inotify` API. The inotify API provides a mechanism for monitoring file system events and allows one to detect accesses to individual files or monitor directories. When a directory is monitored, inotify will return events for the directory itself, and for files inside the directory. To determine what events have occurred, an application reads from the inotify file descriptor. If no events have occurred, the `read` will block (until at least one event occurs). The API allows us to wait for many events: reads, writes, closes, changes in attributes, etc. For our backdoor, we will wait for any read to the obscure file, but we can easily wait for other events. Thus, to wait for a file being accessed, the `inotify` API gives us a file descriptor with the ability to do a blocking read which returns when a file is read. This serves as our trigger.

When the `read` returns, we know that someone has accessed the file, but we cannot be entirely sure that it was our trigger or an unrelated event. To find out, we will spawn a thread that waits for a remote party (the backdoor master) to connect to our socket. If nobody connects, we assume that the file access is unrelated, close the socket and go back to monitoring file accesses. If there is a connection, we spawn a shell.

To accomplish this, the SROP system call chain of our backdoor follows the blocking read of the `inotify` file descriptor with a `clone()` system call (last system call in the leftmost column in Figure 6). This system call has a useful property: using `clone()`, it is possible to assign a different stack to the child process, pointing it to a different state in our automaton. Thus, while the parent resumes waiting for the trigger file, the child will be responsible for the backdoor connection. The actions of the child are shown in the two remaining columns of Figure 6. The child, sets up an `alarm()` clock and listens on a socket, blocking on `accept()`. If no-one connects, this process will be killed by the alarm and we resume monitoring the file accesses of our trigger file. If someone does connect, the alarm is reset and through a series of further system calls, a shell is spawned.

We have implemented the backdoor and tested it on several Linux distributions including the ones mentioned earlier, but also 32 bit variants. As there is no shellcode in memory, the backdoor is very hard to find for a security scanner.

VIII. SROP TO CIRCUMVENT CODE SIGNING

While on Linux it is generally possible to use ROP or SROP to bootstrap more traditional shellcode, other systems like the Apple iPhone's iOS allow only signed code to run natively. Being able to run unsigned code on those systems has become a goal in itself. The jailbreaking community usually uses kernel vulnerabilities to disable the checks that verify these signatures. A well tested method to get control over the kernel is to exploit vulnerable system call interfaces. This, however, does mean that these exploits themselves have to be executed from processes running signed code.

In this section we will describe a technique for a system call proxy using SROP. The technique provides a very generic method of delivering kernel exploits from a process running signed code.

A. Sigreturn on iOS

As a system call proxy is particularly useful for systems requiring signed code, we have implemented this on iOS. When it comes to signaling, iOS (just like Mac OS X) has a small trampoline function from which they indirectly call a signal handler. Upon returning from this signal handler, the trampoline loads the signal frame address from the stack into `r0` (the first system call argument) and then `calls sigreturn`. In iOS we can therefore immediately identify three useful gadgets from the signal handling code:

1) Sigreturn with the signal frame pointer loaded from the stack.
2) Sigreturn with the signal frame pointer in the first function argument.
3) Syscall and return gadget `svc 0 ; bx lr`

The second gadget is useful in case of a function pointer override. We will be using the first and the third gadget for our system call proxy. It is good to note that, unlike on Linux, the iOS/Mac OSX signal frame always contains pointers, so blindly executing signal frames without knowing the location of any data is hard.

B. A system call proxy

The goal of a system call proxy is to remotely control the system calls executed by a process. In our system call proxy automaton (Figure 7) we follow the same basic pattern as described in Section VII of chaining system calls. To bootstrap the automaton, an initial signal frame relocates

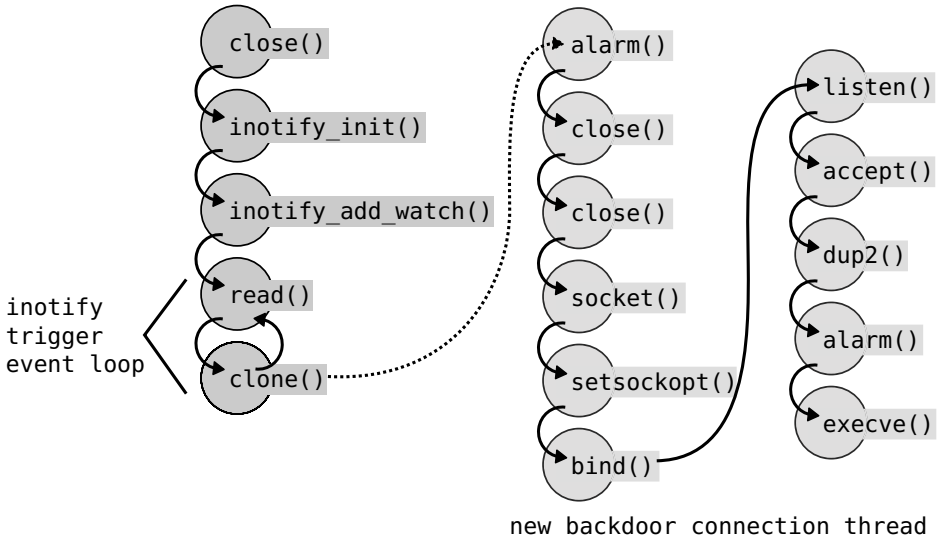

Figure 6. Our example backdoor automaton. The main backdoor thread sets up an `inotify` watch list. When an attacker causes a file to be accessed, a child thread spawns, allowing the attacker to connect to a backdoor shell.

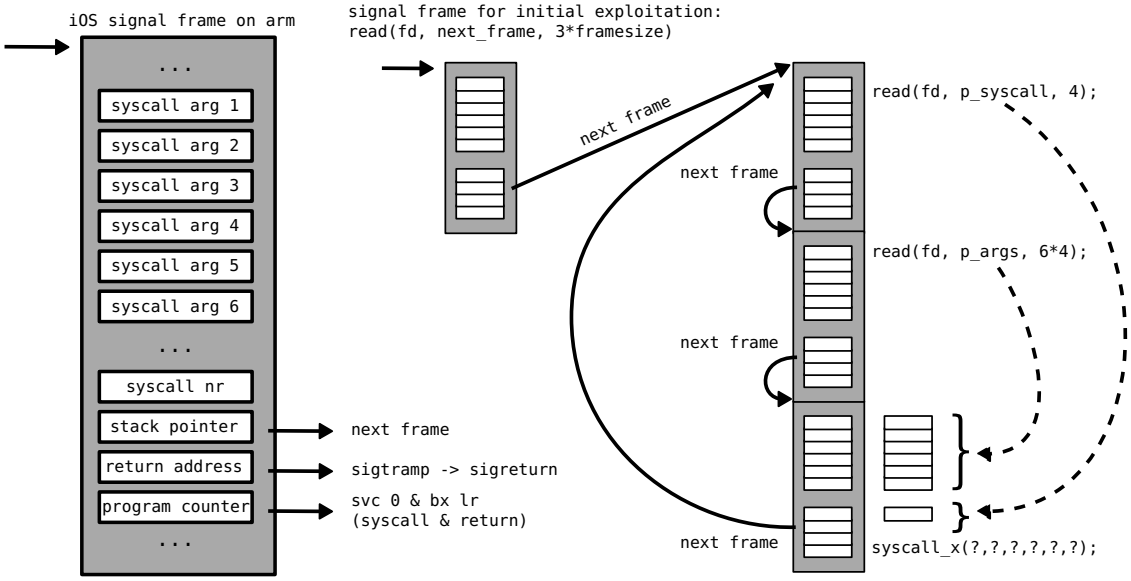

Figure 7. A syscall proxy controllable over a pipe/file/network socket. An initial signal frame sets up a read system call to read in a cyclic automaton which alternates between reading a system call number and its arguments from a socket, and executing said system call.

the stack and issues a `read` system call on a file descriptor controlled by the attacker, this could for example be a network socket, or a file. This `read` loads the automaton onto the new location of the stack. The automaton itself is a loop of one or more signal frames doing `read` system calls and one signal frame that will execute an arbitrary system call. The `read` system calls are responsible for filling in the system call number and their correct arguments in the last signal frame, allowing the attacker to simply supply the whole system call over a socket. System calls that use pointers to data structures as arguments could be preceded with calls to `mmap` and `read`.

IX. The Linux system call interface makes SROP Turing complete

While a simple automaton chaining together system calls using sigreturn is enough for a backdoor to do its job, attackers may want to encode more complex logic such as obfuscation.

It is clear that if we keep the contents of the signal frames in our automaton the same during execution, that the best we can do is execute a static set of system calls, possibly in a loop. But this changes when we allow our automaton to write back into its own signal frames, changing computations to come. In fact, using an automaton which modifies function arguments and stack pointers in future stack frames, we can construct an interpreter for a Turing-complete language.

Our language has a direct mapping to "brainfuck", a well-known Turing-complete language [16]. Conceptually, our language makes use of three registers, the program counter `PC`, the memory pointer `P` and a temporary register used for 8-bit addition `A`. These registers are modeled as file descriptors. The file they have open is `/proc/self/mem`, which on Linux is a way of reading and writing to your own address space. Adding and and subtracting to and from these registers is done using `lseek`. The `PC` file descriptor points to our interpreted language program in memory. Its instructions are addresses of signal frames which implement the following operations:

1) Jump (followed by an offset used by a relative lseek to move the `PC`).
2) Pointer addition/subtraction (followed by an offset used by a relative lseek to move `P`
3) 8-bit Addition/subtraction (followed by a constant to add to the byte located at `P`
4) Conditional jump (the same as Jump, except that it only happens if the byte at `P` is zero).
5) Getchar of the byte at `P`
6) Putchar of the byte at `P`
7) Exit

Figure 8 shows the control flow diagram of the entire state machine. Structurally, it is shaped like a dispatcher, capable of executing the language's commands. Compared to the original brainfuck, our language is a little richer. Instead of increment and decrement, we offer a more generic add of 8 bit numbers.

Instruction dispatch happens by doing a read on `PC`, storing the value in the stack pointer of a dispatch signal frame. When the automaton then proceeds to to a do sigreturn, it will jump to the signal frame belonging to the instruction it has just read.

Reading the immediates following the instructions is also done simply by reading data from `PC`. For pointer addition and for jumps, we use `lseek` with a `SEEK_CUR` argument to move `P` and `PC` respectively.

Conditional jumps are implemented the same way as a normal jump, with the exception that the byte value at `P` is first read into the high byte of the file descriptor argument for the `lseek` on `PC`. If the value at `P` is not 0, the `lseek` will be done not on `PC`, but on a very high, non-existing file descriptor, causing lseek to fail instead of seek, therefore making the jump conditional.

The most complex operation turns out to be our 8 bit addition of data in memory. For this we use a 512 byte buffer, each byte filled with the modulo 256 of its index. For an addition we do an absolute seek with our `A` register to the start of the buffer, followed by 2 relative seeks given by the current value at `P` and the current instruction's immediate, the result of the addition can now be read from `A`.

We have implemented an emulator for [16] which first translates it to our machine language and then proceeds to run it on our automaton.

X. Mitigation

When implementing exploit mitigations, one has to consider the trade-off between performance loss and security gain.

We recognize that sigreturn oriented programming by itself is not an exploitable vulnerability. Similar to ROP, it is an exploitation method that can be used in the event of a vulnerability. Often, ROP may be used instead of SROP and vice versa. Also, some of the 'universal SROP' variants we discovered in Linux on x86-64 and on ARM have been mitigated in recent Linux kernels. On x86-64, the default configuration now uses vsyscall emulation to eliminate useful static gadgets and on ARM, sigreturn has been removed from the static vectors page. However, this does not stop sigreturn oriented programming from being used as a generic stealthy backdoor. Also, if a `syscall & return` gadget can be found inside the binary, using a generic SROP exploit is still easier than using a binary specific ROP chain. We see SROP as a low hanging fruit for exploit writers, worthy of mitigation.

A possible approach to eliminate sigreturn oriented programming as a viable exploitation method would be to embed a kernel-supplied secret value in the sigreturn frame. Upon returning from a signal, the kernel would check this secret value against the value it had written earlier. If the

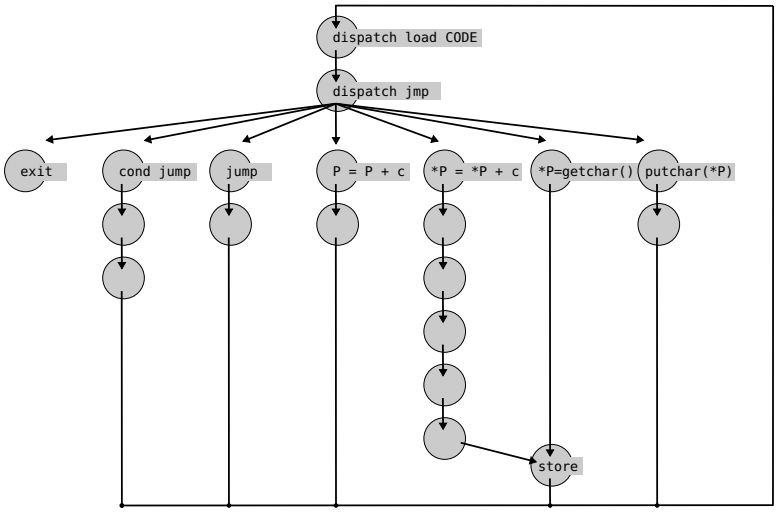

Figure 8. Our Turing-complete interpreter

value is different, the kernel can opt to let the process crash. This method is very similar to stack canaries which have been widely adopted to protect against stack buffer overflows. It also suffers from the same weakness: If an attacker can leak this secret value, he or she can use that to forge fake signal frames. This risk could possibly be remedied by making the kernel zero out the secret value during the sigreturn system call, so that the value is only present in user space while the signal handler is running. The signal canary could even be a cryptographic message authentication code on the complete signal frame, to prevent arbitrary modifications, even in the event of a memory leak.

A complimentary solution could be to keep a counter per process in kernel space which keeps track of the number of signal handlers currently executing. Upon signal delivery, the counter is increased, while a sigreturn decreases the counter. If the counter becomes negative, the process is killed. While this should work fine with single threaded processes, there could be complications with this scheme in multi-threaded programs. Keeping a counter per thread might break programs that use lightweight threads, which may switch user space contexts between threads. This may cause a signal to be delivered in one thread and return in another. On the other hand, keeping a counter for the whole thread group could lead to an overestimation of the number of delivered signals when one of the threads does a `fork()`, unsharing the address space with the other threads. This seems to us as the lesser of two evils.

Ultimately, there's the question whether we can change the behaviour of sigreturn at all without breaking user space applications. Kernels are supposed to have a stable ABI and for that reason it is not done to change the behaviour of system calls. User space programs might break when they depend on previous behaviour. However, sigreturn seems to be a special case. The only legitimate way of calling sigreturn seems to be when user space has been set up by the kernel to call it. Also, depending on the CPU features the signal frame may differ, as for example, SSE and AVX registers will be saved on platforms that support these instruction sets. So, while the location of the general purpose registers seems to be pretty stable and accessible from within a signal handler, being able to manually create a stack frame to return to should in our opinion not be considered part of the ABI.

All things considered, we strongly feel that mitigation against sigreturn oriented programming as an exploitation method is needed.

XI. Conclusion

In this paper, we have discussed sigreturn oriented programming, a novel, Turing complete technique for programming a novel type of weird machine. Sigreturn oriented programming is a generic technique, as we demonstrated by using it for an exploit, a backdoor, and a code-signing bypass. Moreover, it works on a wide variety of operating systems and different architectures. For several of these systems, sigreturn oriented programming permits exploitation without any precise knowledge about the executable. Moreover, the exploit is reusable, as it does not depend much on the victim process at all.

Sigreturn oriented programming represents a convenient, portable technique to program arbitrary code even in strongly protected machines. The number of gadgets needed is minimal and in many systems those gadgets are in a fixed location. As such the technique ranks among the lowest hanging fruit currently available to attackers on UNIX systems. It is important to emphasize that even if kernels are patched to eliminate these fixed-location gadgets, the

usefulness for backdoors is undiminished. In summary, we believe that sigreturn oriented programming is a powerful addition to the attackers' arsenal.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their excellent feedback. This work was supported by the ERC StG project "Rosetta" and by the EU FP7 "SYSSEC" project.

REFERENCES

[1] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th conference on USENIX Security Symposium*, SSYM'05, 2005.

[2] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[3] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[4] Bulba and Kil3r. Bypassing StackGuard and StackShield. http://www.phrack.org/issues.html?issue=56&id=5article, January 2000.

[5] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] 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*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[7] Thomas Dullien. Exploitation and state machines: Programming the 'weird machine', revisited. In *Infiltrate Conference*, Miami, FLA, April 2011.

[8] Brandon Edwards. Dos? then who was phone? (asterisk exploit related to cve-2012-5976). http://blog.exodusintel.com/tag/asterisk-exploit/.

[9] Sergey Bratus Julian Bangert. Page fault liberation army or gained in translation. http://events.ccc.de/congress/2012/Fahrplan/events/5265.en.html.

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

[11] 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*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.

[12] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. 2003.

[13] Vincenzo Lozzo, Tim Kornau, and Ralf-Philipp Weinmann. Everybody be cool, this is a roppery! In *Proceedings of BlackHat USA*, Las Vegas, USA, 2010.

[14] Microsoft. Data execution prevention.

[15] Matt Miller. Preventing the exploitation of seh overwrites. *Uninformed Journal*, 5, 2006.

[16] Urban Müller. Brainfuck–an eight-instruction turing-complete programming language. *Available at the Internet address http://www.muppetlabs.com/ breadbox/bf/*, 1993.

[17] 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 (ACSAC)*, ACSAC '10, pages 49–58. ACM, December 2010.

[18] Vasilis Pappas. kbouncer: EïňČcient and transparent rop mitigation. In *Usenix Security, Microsoft BlueHat Prize winner.*, 2013.

[19] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[20] PaX Project. Address Space Layout Randomization. http://pax.grsecurity.net/docs/aslr.txt, 2001.

[21] Dennis Ritchie. Unix version 6 – signal.s. http://minnie.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/s5/signal.s, 1975.

[22] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[24] 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*, CCS'07, 2007.

[25] 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 (CCS'04)*, 2004.

[26] Rebecca Shapiro. The care and feeding of weird machines found in executable metadata. http://events.ccc.de/congress/2012/Fahrplan/events/5195.en.html.

[27] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[28] Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63, August 1997.

[29] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[30] Julien Vanegue. The automated exploitation grand challenge, tales of weird machines. In *H2C2 conference*, Sao Paulo, Brazil, October 2013.

[31] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: when benign apps become evil. In *Proceedings of the 22nd USENIX conference on Security*, SEC'13, pages 559–572, Berkeley, CA, USA, 2013. USENIX Association.

[32] Michal Zalewski. "delivering signals for fun and profit". http://lcamtuf.coredump.cx/signals.txt, 2001.