Escape From Return-Oriented Programming: Return-oriented Programming without Returns (on the x86)

Stephen Checkoway
UC San Diego

Hovav Shacham
UC San Diego

Abstract

We show that on the x86 it is possible to mount a return-oriented programming attack without using any return instructions. Our new attack instead makes use of certain instruction sequences that behave like a return; we show that these sequences occur with sufficient frequency in large Linux libraries to allow creation of a Turing-complete gadget set.

Because it does not make use of return instructions, our new attack has negative implications for two recently proposed classes of defense against return oriented programming: those that detect the too-frequent use of returns in the instruction stream, and those that detect violations of the last-in, first-out invariant that is normally maintained for the return-address stack.

1 Introduction

This paper is about defenses against return-oriented programming.

Return-oriented programming. Return-oriented programming allows an attacker to exploit memory errors in a program without injecting new code into the program's address space. In a return-oriented attack, the attacker arranges for short sequences of instructions in the target program to be executed, one sequence after another. Through a choice of these sequences and their arrangement, the attacker can induce arbitrary (Turing-complete) behavior in the target program. Traditionally, the instruction sequences are chosen so that each ends in a "return" instruction, which, if the attacker has control of the stack, allows control to flow from one sequence to the next — and gives return-oriented programming its name.

The organizational unit of return-oriented programming is the *gadget*, an arrangement of instruction sequence addresses and data that, when run, induces some well-defined behavior, such as xor or an unconditional jump. Return-oriented exploits begin by devising a Turing-complete gadget set, from which any desired attack functionality is then synthesized.

Return-oriented programming was introduced by Shacham in 2007 [24] for the x86 architecture. It was subsequently extended to the SPARC [2], Atmel AVR [10], PowerPC [18], Z80 [3], and ARM [17] processors. While the original attack was largely manual, later work showed that each stage of the attack can be automated [2, 22, 14, 17].

Defenses against return-oriented programming. The instruction stream executed during a return-oriented attack as described above is different from the instruction stream executed by legitimate programs in at least two ways: first, it contains many return instructions, just a few instructions apart; second, it unwinds the stack with return instructions for which there were no corresponding "call" instructions. These two differences have been proposed by researchers as a way of detecting and defeating return-oriented attacks:

- The first difference suggests a defense that looks for instruction streams with frequent returns. Davi, Sadeghi, and Winandy [7] and Chen et al. [4] both use dynamic binary instrumentation frameworks

(Pin [20] and Valgrind [21], respectively) to instrument program code. With both systems, three consecutive sequences of five or fewer instructions ending in a return trigger an alarm.

- The second difference suggests a defense that looks for violations of the last-in, first-out invariant of the stack data structure that the call and return instructions usually maintain in benign programs. Buchanan et al. [2] suggest that the shadow return-address stack maintained by the SPARC-specific StackGhost system [12] can be used to defend against return-oriented programming. Francillon, Perito, and Castelluccia [11] implement a shadow return-address stack in hardware for an Atmel AVR microcontroller; only call and return instructions can modify the return-address stack.

Our contribution. We show that, on the x86, it is possible to perform return-oriented programming *without using return instructions*. We show that instruction sequences exist that behave like a return, and that these can be used instead of returns to chain useful instruction sequences together to produce Turing-complete functionality. The particular return-like instruction sequences we use are of the form "pop x; jmp $*x$", where x is any general-purpose register, though we speculate that other kinds of return-like sequences may be usable for return-oriented programming. We discuss our techniques for using such sequences in place of returns in Section 2.

Although these sequences are less frequent than returns, certain incidental characteristics of the x86 instruction set architecture (ISA) make them sufficiently frequent in large libraries to use in attacks; we discuss this in Section 3. In Section 4 we describe a Turing-complete gadget set we have created based on the libc and certain large libraries distributed with Debian GNU/Linux 5.0.4 ("Lenny"). For certain classes of memory errors — notably, for setjmp buffer overwrites — it is possible for an attacker to take over the program's control flow without executing even one return. For other classes of memory errors, a single overwritten return address is needed, after which no further returns are executed. We discuss this in Section 5. For completeness we give, in Section 6, a complete return-oriented exploit without return instructions against a sample target program.

Negative implications for defenses. Our attack has negative implications for defenses against return-oriented programming that look for return instructions in order to recognize a return-oriented instruction stream. Defenses of the first kind considered above, which detect the use of several return instructions in close succession, will not detect attacks structured like the ones we introduce in this paper since these attacks make use of either one return or none at all. When it is possible to initiate an attack without a return the LIFO invariant of the return-address stack is not violated, so defenses of the second sort will also not detect the attacks.

Because our attack does not violate the LIFO invariant of the return-address stack, it is not clear that defenses of the second kind (which maintain a shadow return-address stack) can be salvaged. Maintaining a shadow copy of jump targets would not be useful, because no simple invariant governs these targets in benign programs.

On the other hand, it may be possible to patch defenses of the first kind to look not just for several returns in quick succession but also for several indirect jumps in quick succession. This would detect attacks structured as ours are. Doing so without being able, provably, to detect that every kind of return-like instruction sequence that a return-oriented program might use risks engaging in a classic cat-and-mouse game in which attackers switch to new return-like sequences to evade the upgraded defenses. Prior to our results in this paper, it appeared that return-oriented programming unavoidably relied on return instructions, making these instructions attractive targets for detection and defense. Now, however, it appears that a different property must be found by which to detect return-oriented attacks.

2 Return-Oriented Programming without Returns

In this section we describe how return-like instruction sequences can substitute for rets, allowing return-oriented programming without use of return instructions.

2.1 Return-like instruction sequences

A ret instruction has the following effects: (1) it retrieves the four-byte value at the top of the stack, and sets the instruction pointer (eip) to that value, so that the instructions beginning at that address execute; and (2) it increases the value of the stack pointer (esp) by four, so that the top of the stack is now the word above the word assigned to eip. This is useful for chaining return-oriented instruction sequences because the location of each sequence can be written to the stack; when an instruction sequence has executed, reaching the ret that ends it, that ret causes the next instruction sequence to be executed.

One way to view this arrangement, suggested by Roemer et al. [23], is that in return-oriented programming the stack pointer takes the place of the instruction pointer in ordinary programming; that each gadget on the stack is an instruction for a custom-built virtual machine; and that the ret at the end of each instruction sequence acts like a typewriter carriage return to advance the processor to the next instruction — something the processor does automatically for ordinary programs.

Consider the following instruction sequence

pop %eax; jmp ∗%eax.

This sequence behaves like a ret in inducing effects (1) and (2) above. Its only side effect is in overwriting the former contents of the eax register. The pop %eax; jmp ∗%eax sequence is return-like. The set of instruction sequences in a target program that end in pop %eax; jmp ∗%eax — provided they do not make use of eax for dataflow — can be chained together for return-oriented programming just as if they had ended in a ret instruction. This is the central observation of this paper.

In fact, there are many more return-like instruction sequences that can be used besides pop %eax; jmp ∗%eax. First, any of the other general-purpose registers (esp excepted, for obvious reasons) can be used in place of eax. Second, just because ret sets eip to the value at the top of the stack there is no reason that all return-like instruction sequences must. For example, the sequence pop %eax; jmp ∗(%eax) uses a doubly indirect jump to set eip to the value contained in the memory word pointed to by eax. If the attacker wishes eip to take the value x, she simply picks some other memory location y, stores x there, and places the value y at the top of the stack, where the pop instruction assigns it to eax. Since the attacker controls the stack, this is no harder for her than storing the value x at the top of the stack for ordinary ret instructions. A return-oriented exploit that uses such doubly indirect jumps can be organized to include a *sequence catalog* of useful instruction sequence addresses, something like the Global Offset Table used in dynamic linking. (As before, any other general-purpose register can substitute for eax in the pop %eax; jmp ∗(%eax) sequence.)

What's more, a doubly indirect jump with an immediate offset (either 8-bit or 32-bit) is just as useful as one without an offset. To use the sequence pop %eax; jmp ∗c(%eax), where c is some constant, the attacker must simply store not y on the stack but $y - c$. Once more, any register can substitute for eax.

Finally, there are two kinds of doubly indirect jumps on the x86: near and far. A near jump takes a 32-bit address in the current segment; a far jump takes a 32-bit address together with a 16-bit segment selector. Far jumps allow for sophisticated privilege domain regimes with restricted cross-domain calls (they are used, for example, in the Native Client sandbox [26]). For our purposes, however, we need only the following fact: An appropriate choice of segment selector (on our Debian system, `0x0073`) leaves the code segment unchanged; a far jump to an address with this segment selector behaves exactly like a near jump to the same

address.[1] Because the segment selector follows the address in memory, we can follow each address in the sequence catalog with the appropriate segment selector and thereafter use far and near doubly indirect jumps interchangeably. (This introduces zero bytes into the catalog; if this is a problem for a particular exploit, the zero bytes can be patched in at runtime; see Section 6.)

We use all the pop-jump sequences described above in constructing our gadgets. For brevity, we refer to all of them using the shorthand pop *x*; jmp ∗*x*, where *x* refers to any general purpose register. The jump may be indirect or doubly indirect; and, if doubly indirect, it may be near or far, and it may take an 8- or 32-bit immediate offset.

Other types of return-like sequences. More generally, there are two crucial features of ret that return-like instruction sequences must emulate: ret transfers control to some new instruction sequence; and it changes some global state so that a second ret transfers control to a different instruction sequence (rather than inducing an infinite loop). Like ret, the instruction sequences we describe above, and which we use in building our Turing-complete gadget set, change global state by increasing esp by four. But this is not an absolute requirement. One could imagine an instruction sequence based on call ∗*x*, which would *decrease* esp each time it is used. Or a different register could be used, as, e.g., in jmp ∗(%eax); add `0x4`, %eax. Or, using SIB addressing, a combination of registers could be used, with the index register scaled by 4 and incremented after each dereference. Or a memory location could serve as the mutable state instead of a register. The point here is that many possible types of instruction sequence have return-like behavior and are potentially suitable for return-oriented programming. A defense that detects some but not all of these types of instruction sequences would be of limited value, as attackers may be able to switch to a different return-like sequence and thereby evade detection.

2.2 Reusing a pop-jump sequence

As shown above, a pop *x*; jmp ∗*x* sequence can be used in place of a ret instruction in return-oriented programming. One way to create a return-oriented attack without returns is to look, in the target binary and the libraries it links against, for instruction sequences ending in pop *x*; jmp ∗*x* (for various registers *x*), then choose from among those sequences to construct gadgets.

As we show in Section 3, properties of the x86 ISA mean that pop *x*; jmp ∗*x* sequences occur not infrequently in large programs. But they are still not common. For example, our two test libcs happen to include only a single usable pop *x*; jmp ∗*x* between them. If there are only a few pop *x*; jmp ∗*x* sequences then there are only a few sequences ending in pop *x*; jmp ∗*x*. And if only these sequences are useful for an attacker in constructing a return-oriented attack, then she may need a very large amount of code in the target program to find sequences sufficient for achieving Turing completeness.

But in fact there is no need for every instruction sequence to end in pop *x*; jmp ∗*x*. Shacham observed [24, Section 5.1] that if ebx contains the address of a ret instruction then any instruction sequence ending in jmp ∗%ebx behaves just as if it had ended in ret; the same is true for other registers and for doubly indirect jumps of various kinds.[2]

The crucial point is that this equivalence holds true even if ebx contains the address not of an actual ret but of a return-like instruction sequence. Suppose the target of jmp ∗*y* is a pop *x*; jmp ∗*x* sequence (where

[1] A 16-bit segment selector consists of a 13-bit index, a 1-bit table indicator, and a 2-bit requested privilege level. The index specifies a 64-bit segment descriptor in either the global descriptor table or the local descriptor table as specified by the table indicator. Each segment descriptor contains a number of bit-fields including the segment base address, segment limit and privilege level. Since Linux uses a flat address space, most of the segment descriptors used in user programs specify a base address of zero and a limit of 4 GB [16]. The selector `0x0073` corresponds to an index of 14 in the global descriptor table with a requested privilege level of ring 3.

[2] Cf. [6, 19, 5] for the use of similar techniques in the context of code injection.

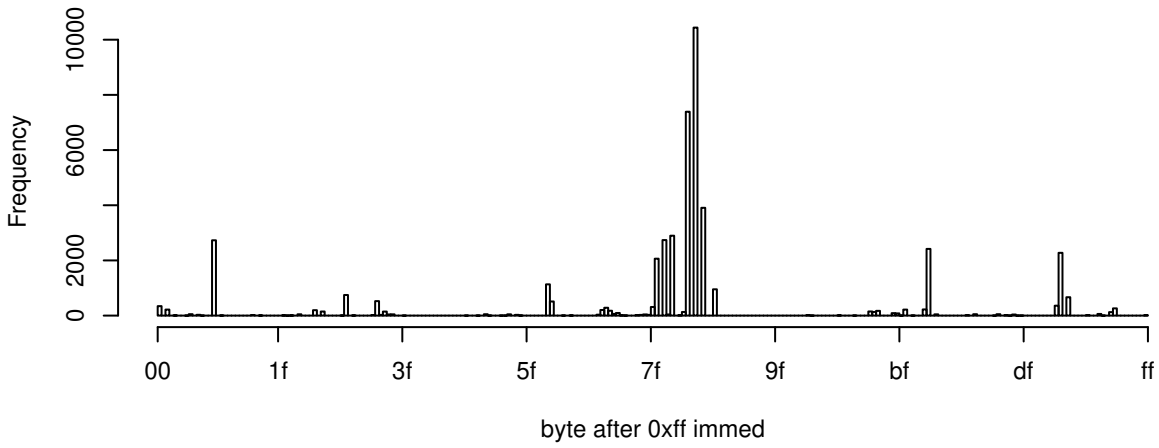

Figure 1: Distribution of byte values following `ff` immediate byte, in libc from Debian 5.0.4 ("Lenny").

x and y refer to different registers). Then any instruction sequence ending in jmp $*y$ will behave just as if it had ended in ret (except, again, that the value in the x register is overwritten).

It is not necessary that all sequences use the same register in their jmp $*y$ instruction: it is easy to load immediate values into registers (using pop or popad), so the pop x; jmp $*x$ address can be made the target of whatever register is required for a particular instruction sequence. Thus any sequence ending in jmp $*y$ (where y refers to any general-purpose register) is useful for return-oriented programming. There are many more such sequences than only those ending in a pop x; jmp $*x$ sequence, which means that Turing completeness can be obtained from smaller target programs.

3 The Availability of Pop-Jump Sequences

Whereas traditional return-oriented programming relies on the availability of diverse and useful instruction sequences ending in a ret instruction, our new return-oriented programming relies on, first, the availability of return-like pop-jump sequences of the form pop x; jmp $*x$; and, second, the availability of diverse and useful instruction sequences ending in jmp $*x$. In this section, we consider whether such sequences will occur often enough to make construction Turing-complete gadget sets possible.

On the x86, the return instruction is a single byte, `c3`, which we would expect to occur with frequency $1/256$ in a random byte stream, and which in fact is even more frequent in machine code because legitimate programs regularly use ret.[3] By contrast, indirect jumps through a registers are *two bytes* on the x86, and these instructions are also less frequently used in legitimate programs than are rets. It is not *a priori* clear that sufficiently many jmp $*x$ instructions will exist in a target program, or that they will be preceded by diverse and useful other instructions.

Here an incidental characteristic of the x86 ISA comes to our help. The first byte of all indirect jumps (both near and far) is `ff`. What's more, many x86 instructions include immediate values; immediate values are encoded last in any instruction that includes an immediate; and immediate values, like other numbers, are encoded in two's complement, little endian. Thus the last byte of every instruction that includes an immediate value that is negative and in the range -1 to -16777216 will be `ff`. Such immediate values are very common. Out of the 83554 four-byte immediate values in instructions in our test libc, 46530, or 55%, have last byte `ff`. (Another 36369 have last byte `00`.)

[3] In fact, the x86 includes at least four different usable return instructions, each just a single byte.

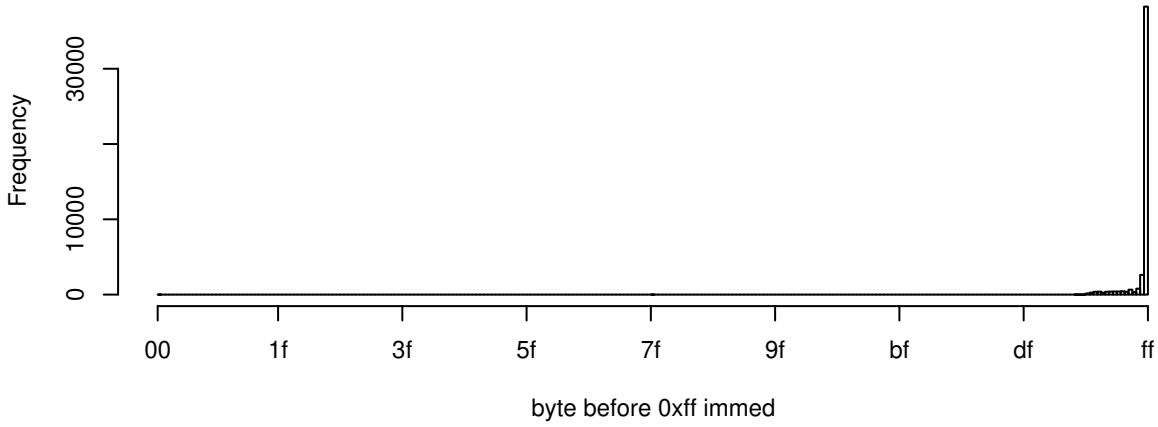

Figure 2: Distribution of byte values preceding `ff` immediate byte, in libc from Debian 5.0.4 ("Lenny").

One way to obtain jump instructions, then, is to take the opcode byte (`ff`) from the last byte of the immediate value in a legitimate instruction in the target binary. Because this byte is the very last byte in the encoding of that first instruction, the second byte of our jump will coincide with the first byte of the next legitimate instruction in the target binary. We thus require that this instruction's opcode be some value that, as a second byte following `ff`, is one that specifies a useful jump. Figure 1 shows the distribution of bytes immediately after such `ff` bytes in our test libc. The two most common bytes are `8b` (10439 occurrences) and `89` (7389 occurrences), both forms of mov (these are opcodes for, essentially, store and load instructions, respectively). When interpreted as a byte following `ff`, sadly, neither of these, specifies a jump. (Both are kinds of `ff/1`, which is the decrement long instruction.)

Out of the 256 possible values for the second byte, 56 encode indirect jumps: `20–2f`, `60–6f`, `a0–af`, and `e0–e7`.[4] In the distribution of bytes we see immediately after a most-significant immediate `ff` byte, `66` (gs segment override, 1113 occurrences) and `65` (operand size override, 511 occurrences) are particularly frequent. There is thus enough diversity in bytes following an `ff` immediate that jmp $*x$ instructions are available.

The fact that the last byte of an immediate value and the first byte of the following instruction frequently makes a jump instruction would not be of value to us if that instruction were not preceded by other useful instructions. Here again an incidental characteristic of the ISA is of help: In many cases, the byte before the jump instruction is essentially a one-byte no-op, and the bytes before that no-op vary greatly. Figure 2 shows the distribution of second most significant bytes in immediates whose most significant byte is `ff`, again in our test libc. Not surprisingly, these values are mostly `ff` or close, meaning they encode small negative numbers. Although `ff` and `fe` encode two-byte instructions,[5] `fd` and `fc` encode std and cld, which respectively set and clear the direction flag. The direction flag governs the behavior of string instructions, and its value is irrelevant for the behavior of the gadgets we construct. As libraries become larger, the likelihood that offsets encoded as immediates will be in the range -131073 to -262144 (that is, will have more significant half `fc ff` or `fd ff`, in little-endian) increases.

Compared to jmp $*x$ instruction, pop x sequences are more frequent. A pop into each general-purpose register has its own one-byte instruction, from `58` (pop %eax) to `5f` (pop %edi).

[4]The values `e8–ef` encode far indirect jumps of the form "ljmp $*$%eax" or another register and are invalid instructions, since far jumps target m16:32 [15].

[5]Including, as we have observed, indirect jumps.

Putting everything together, we see that incidental features of the x86 ISA mean that instruction sequences ending "std; jmp $*x$" and "cld; jmp $*x$" are quite common in large libraries. Many of the instruction sequences we use to construct our Turing complete gadget set in Section 4 are of this form.

Of course, `ff` as the last byte of an immediate value is not our only source of jump instructions. We are able to use legitimate indirect jumps in the target binary, and `ff` bytes can also occur as ModR/M bytes, SIB bytes, or as other parts of an immediate value. We focus on most-significant immediate `ff` bytes because the jump instructions they engender arise naturally from properties of the x86 ISA, and would thus be difficult to eliminate by changing the compiler.

4 A Gadget Catalog

To demonstrate that Turing-complete return-oriented computation without returns is feasible in real programs, we design a set of *gadgets* each of which performs a discrete computation and can be reasoned about independently by virtue of little or no state maintained between gadgets. We build these gadgets by looking at the C standard library found in Debian GNU/Linux 5.0.4 ("Lenny"), GNU libc 2.7, which is 1294572 bytes.[6] As we will see below, by itself, Debian's libc is almost sufficient. We need a single instruction sequence to exist in the either target program or in a library loaded by the target program. We find this additional instruction sequence in two large libraries: Mozilla's libxul (11857460 bytes), distributed with Firefox and Thunderbird; and the PHP language's libphp5 (5450680 bytes). These libraries are, of course, used in Web browsers and Web servers, respectively, which make common targets for exploitation.

As described in Section 2, rather than using sequences of instructions that end in pop x; jmp $*x$, we use sequences of instructions that end in jmp $*y$ where y is a pointer to a pop x; jmp $*x$ sequence. It is exactly this pop x; jmp $*x$ that we do not find in libc[7] and so must exist in the target program or one of its libraries. We call this (facetiously) the *bring your own pop-jump* (BYOPJ) paradigm.

Because libc is loaded into every Linux executable, we gain confidence by using it as the corpus for our instruction sequences (except the pop-jump) that return-oriented programming without returns is likely possible in any large Linux program that an attacker might target. We stress that using most instruction sequences from libc but a pop-jump from libxul is not how a real attacker would go about mounting an attack. Libxul is larger and has more convenient instruction sequences than libc does; a Turing-complete gadget set could be constructed more easily from libxul alone than from libc with a libxul pop-jump. However, any program that did not link against libxul would require an entirely different gadget set. Unlike creating a new gadget set, testing that a program contains a suitable pop-jump is simple and easily automated.

Most of the useful instruction sequences end with either a near (resp. far) indirect jump to the address stored in the near (resp. far) pointer in memory at an address stored in register edx. That is, many instruction sequences end with jmp $*$(%edx) or ljmp $*$(%edx).

Each gadget could be made fully independent from the others, but since register edx is so useful for chaining instruction sequences, we ensure that at the end of each gadget, it holds the address of the sequence catalog entry for the pop x; jmp $*x$. In most cases, this required no additional work. The function call gadget is the only one which required the fix up.

[6] There are actually two distinct libcs on our test system: `/lib/libc-2.7.so` and `/lib/i686/cmov/libc-2.7.so`. The gadgets described in this section and the example exploit in Section 6 are constructed from the former. However, the latter library is loaded at runtime instead on some machines, apparently those that support the conditional-move instructions cmov*cc* (introduced with the Intel Pentium Pro). We have verified that this libc also provides instruction sequences sufficient for constructing a Turing-complete gadget set without returns. (As it happens, the most convenient way of constructing gadgets from instruction sequences in this library more closely resembles Shacham's original gadget set [24] than the set described in this section.) That either one of these libcs suffices for obtaining Turing-complete return-oriented programming without returns gives strong evidence for our thesis in this paper.

[7] In the second libc described in footnote 6, there is a single pop %edx; jmp $*$(%edx) sequence but as we show below, edx is too useful to use for this purpose. Other minor differences exist between the two libraries but we do not dwell on them further.

Following Checkoway et al. [3], we design a three-address code collection of memory-memory gadgets — that is, our gadgets are of the form $x \leftarrow y \,\mathrm{op}\, z$, where x, y, and z are literal locations in memory that hold the operands and destination. As mentioned, we use register edx to chain our instruction sequences and for the pop x; jmp $*x$ sequence in our BYOPJ paradigm, we use register ebx. This means that we cannot store any state in register ebx, but we need not worry about changing its contents during the course of an instruction sequence since it will be overwritten during the pop %ebx. This leaves us with five registers, eax, ecx, ebp, esi, and edi, to do with as we please.

Instruction sequences. We used 34 distinct instruction sequences ending with jmp $*x$ to construct 19 general purpose gadgets: load immediate, move, load, store, add, add immediate, subtract, negate, and, and immediate, or, or immediate, xor, xor immediate, complement, branch unconditional, branch conditional, set less than, and function call. The majority of the instruction sequences contain four or fewer instructions. The sequences were chosen by hand out of a collection of potential instruction sequences in libc discovered by the algorithm given by Shacham [24].

Loading data from the stack into a register can be accomplished by means of a pop x; jmp $*y$ instruction sequence:

```
pop %eax;   sub %dh, %bl;     jmp *(%edx)
pop %ecx;   cmp %dh, %dh;     jmp *(%edx)
pop %ebp;   or $0xF3, %al;    jmp *(%edx)
pop %esi;   or $0xF3, %al;    jmp *(%edx)
pop %edi;   cmp %bl, %dl;     jmp *(%edx)
pop %esp;   or %edi, %esi;    jmp *(%eax)
popad;      cld;              ljmp *(%edx)
```

The first five can be used to load any of the registers we wish to use as long as we load register eax after registers ebp and esi. The sixth allows for a simple jump by changing the stack pointer, see below. Instruction popad pops all seven general purpose registers off of the stack (it does not pop register esp, but it does require 4 bytes on the stack which are ignored for a total of 32 bytes popped off of the stack). Without a pop %edx; jmp $*x$ instruction in the target binary or its libraries, popad is the only way to load register edx. This is only an issue for our function call gadget described below.

The gadgets need to be able to move data between memory and registers as well as between multiple registers. Moving a word from memory into a register is accomplished by means of a mov $n(x)$, y instruction where n is some immediate offset. The analogous instruction mov x, $n(y)$ allows for the reverse operation. Movement between registers is less straight-forward because while such an x86 instruction exists, we find none in sequences ending in jmp $*x$. Instead, the contents of two registers can be exchanged with the xchg instruction, or by arranging for the destination register to be `0x00000000` or `0xffffffff`, the source register can be ored or anded with the destination, effecting the move.

One tricky aspect of return-oriented programming using pop x; jmp $*x$ instead of a return is that we frequently need to use a register for holding data in one instruction sequence as well as for being the x in the jmp $*x$ in another sequence in a single gadget. Handling this requires careful structuring of the instruction sequences inside the gadget to ensure that the register has been loaded with the address of the pointer to the pop x; jmp $*x$ sequence before it is needed.

By now, the gadget-construction procedure is well-described in the literature [24, 2, 10, 14, 13, 18]. As such, we only briefly describe each of our standard gadgets and focus more on the gadgets that require extra finesse.

Data movement. The first thing we wish to do is to *load immediate* values into memory at a fixed address. This is easily accomplished by loading esi with the immediate value and eax with the fixed address plus `0xb`. This takes two pops. Then we use mov %esi, `-0xb`(%eax) to write the immediate value to memory.

Since we want a collection of memory-memory gadgets, we need to load a word from one (constant) location in memory and store it into another (constant) location in memory. This is accomplished by loading the source address into eax, loading the destination address into ebp, loading from eax into edi, and finally storing edi into memory at the address in ebp. This is the *move* gadget.

A simple modification to the move gadget yield the *load* gadget. Rather than storing the word in memory at the source address into the destination address, that word is used as a pointer to another word in memory which is loaded into another register and then stored at the destination address. In pseudo code, the operation is the following.

```
eax ← source
edi ← (eax)
esi ← (edi)
eax ← destination
(eax) ← esi
```

A *store* gadget is similar except that the address where the source value is to be stored is itself stored at a fixed location. That is, the store gadget performs the operation $(A) \leftarrow B$ where A is the word in memory at the destination address and B is the word in memory at the source address. In fact, we can perform the operation $(A+n) \leftarrow B$ where n is a literal value. This allows for easy constant array indexing into an array that is not at a fixed location in memory, where A is the address of the array and n is the offset into the array.

Arithmetic operations. The *add*, *add immediate*, and *subtract* gadgets are straight forward. They work by loading the source operands into registers, performing the appropriate operation, and then storing the result back to memory. The x86 ISA allows one of the operands to be a location in memory which would obviate the need to load one of the operands. This could potentially simplify the gadgets.

The *negate* gadget, loads the word from the source address, takes the two's complement of the word and stores it back to memory. There is an x86 instruction neg that performs the two's complement of a register, but it does not appear near a jmp $*x$ instruction. Instead, we load esi with zero, for example by using xor %esi, %esi and then use the sequence subl `-0x7D`(%ebp,%ecx), %esi; jmp *(%ecx) to subtract the value from zero. The subl instruction performs the operation esi $\leftarrow$ esi $-$ (ebp $+$ ecx $-$ `0x7D`).[8] Since our jmp $*x$ uses ecx, we have to load it with the address of a pointer to the pop x; jmp $*x$ sequence. This means that ebp must have the value of the source address plus `0x7D` minus the address of the pointer to pop x; jmp $*x$.

Logical operations. The *and*, *and immediate*, *or*, and *or immediate* gadgets are constructed in an analogous manner to the add gadget. Namely, the operands are loaded into registers, the operation is performed, and the result is stored back to memory. The only tricky part is the movement of data between registers as described above.

The *xor* and *xor immediate* gadgets are similar except that instead of xoring the value of two registers and then storing the results back to memory, the first source word is written to the destination and that location is subsequently xored with the second source word.

The *complement* gadget stores the one's complement of the source value into the destination address. Similar to the situation with the negate gadget, there is an x86 instruction not which performs the one's complement, but it does not appear in the useful instructions sequences in libc. Instead, we proceed exactly

[8] The parentheses denote dereference, not grouping.

as for the negate gadget except instead of loading esi with zero, we load it with `0xffffffff` $= -1$. This works because $-1 - x = \neg x$.

Branching. In a normal program, there are two ways to perform a branch. The branch can be to an absolute address or to an address relative to the current instruction. In return-oriented programming, a branch is performed by changing the stack pointer rather than the instruction pointer. An absolute branch can be effected by popping a value off the stack into esp. Alternatively, a negative offset from the end of the gadget can be popped into edi which is then subtracted from the stack pointer using the sequence sub %edi, %esp; ljmp $*$(%eax) This allows stack-pointer-relative branching. This is the basis for our *branch unconditional* gadget.

In order to have Turing-complete behavior, we must have a way to perform a conditional branch. The x86 has a number of conditional branch operations; however, these are unsuitable for our purpose since they affect the instruction pointer rather than the stack pointer. Instead, we need a way to change the stack pointer conditioned on the word stored in memory at a known address. If the word is zero, then we do not change the stack pointer. If the word is `0xffffffff`, then we subtract an offset from the stack pointer as in the unconditional case. The way we do this is by loading the word into a register and anding with the offset. The result is subtracted from the stack pointer. The implementation is a straight-forward combination of the and gadget and the branch unconditional gadget and is our *branch conditional* gadget.

In any collection of return-oriented gadgets, the most difficult to construct is the gadget that compares two values and performs an operation based on the relative magnitude of the values. Taking a cue from the MIPS architecture, we implement a *set less than* gadget that sets the word at the destination address equal to `0xffffffff` if the first source word is less than the second source word.

The implementation of the set less than gadget is given in Figure 3. The string compare instruction cmpsl compares the two words pointed to by %ds:%esi and %es:%edi and sets the carry flag if the latter is greater than the former. As a side effect, it increments or decrements registers esi and edi based on the direction flag; however, this is of no concern since we are only comparing a single word. The sbb instruction subtracts esi plus the value of the carry flag from esi. In essence, if the first source value is less than the second source value, then the carry flag will be set and esi is set to `0xffffffff`, otherwise, the carry flag will not be set and so esi will be set to zero, exactly as required for the branch conditional gadget. The one thing we have to be careful of is register cl cannot be zero otherwise a divide by zero exception will occur.

With the set less than and logical gadgets, a conditional branch based on comparing any two values for any of the six relations $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ can be formed. At this point our set of gadgets is Turing-complete.

Function calls. Now that we have a Turing-complete set of gadgets, we extend their functionality by adding a gadget to perform function calls. This gives us two new abilities: we can call normal return-oriented instruction sequences — i.e., those ending in return — or we can call legitimate functions. Since we use an actual call instruction, any return-oriented programming defense relying on the LIFO nature of the call stack will be thwarted since this invariant is maintained. Any defense relying on the frequency of return instructions will be thwarted as long as the number of other instructions executed between these calls is sufficiently high.

Since calling legitimate functions is the more complicated of the two operations, we focus on it here. Calling a sequence ending in return is roughly the same except for moving the stack pointer and handling the return value.

Before a function call is made, the stack pointer must be moved to a new location to keep from overwriting our previous gadgets on the stack. If n is the address where the stack pointer should be when the function begins to execute — i.e., the location where the return address will be stored — then the k arguments should

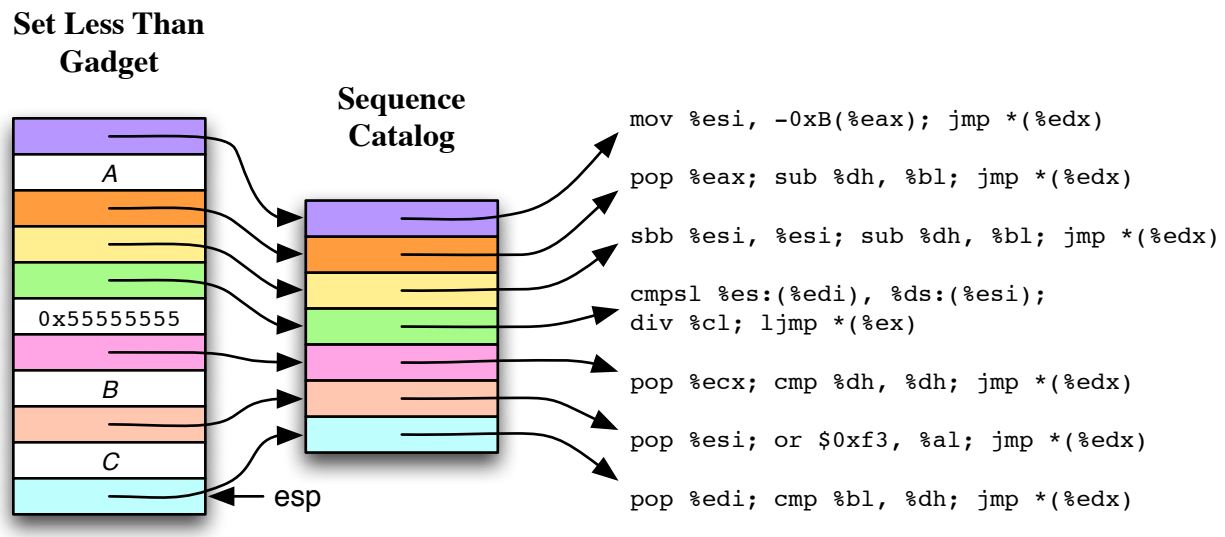

Figure 3: Set less than gadget. If the word at address *B* is less than the word at address *C*, then set the word at address *A* to `0xffffffff`, otherwise set it to `0x00000000`. The gadget begins executing with the stack pointer (esp) pointing to the bottom-most (smallest address) cell of the gadget. As execution proceeds, the stack pointer moves to higher cells (higher addresses). Each cell is either a pointer to an entry in the sequence catalog — which is itself a pointer to the instruction sequence that is actually executed — or data. After the final instruction sequence in the gadget has executed, the stack pointer points to the next gadget to be executed.

be stored at addresses $n+4$, $n+8$, ..., $n+4k$. This can be done using the load immediate or move gadgets. The *function call* gadget is then used to perform the computation $A \leftarrow fun(arg_1, arg_2, \ldots, arg_k)$ with the stack pointer set to n.

Since the Linux *application binary interface* (ABI) for x86 specifies that registers eax, ecx, and edx are caller-saved while registers ebx, ebp, esi, and edi are callee-saved, some care must be taken to ensure that after the function has returned, the gadgets can retain control.

One particularly tricky point is that since edx is caller-saved, once we return from the call we need to restore it to the address of the pointer to the pop *x*; jmp $*x$. We *cannot* do this using only the instruction sequences in libc if we care about the return value which is in eax. Continuing our BYOPJ paradigm, if the target program has either a pop %edx; jmp $*$(%edx) or a pop %edx; jmp $*$(%esi), then we can restore edx without overwriting the return value in eax. Mozilla's libxul has such a sequence. Without such a sequence, the function call gadget has to be tailored for each application rather than being generic.

The implementation of the function call gadget is given in Figure 4. Some parts of the implementation are rather subtle. The first thing it does is to load registers esi, ebp, and eax. Register esi is loaded with the address of the sequence catalog entry for the call-jump sequence, ebp is loaded with the actual address of the leave-jump sequence, and eax is loaded with the literal value n (plus the offset for our store sequence). Next, the address of the sequence catalog entry for the call-jump is stored at address n. Register esi is then loaded with `0x38` and the value of the stack pointer is added to is. At this point, esi holds the address we will set the stack pointer to after the the function call returns.

Now that we know the location on the stack we wish to return to after our function call, we need to move it into ebp. Unfortunately, the easiest way to do that is to store it to memory (at the location where we will eventually store the function's return value), load it back from memory into edi and then exchange it with ebp. After the exchange, edi holds the address of the leave-jump sequence and ebp holds the value we will set the stack pointer to after the function call.

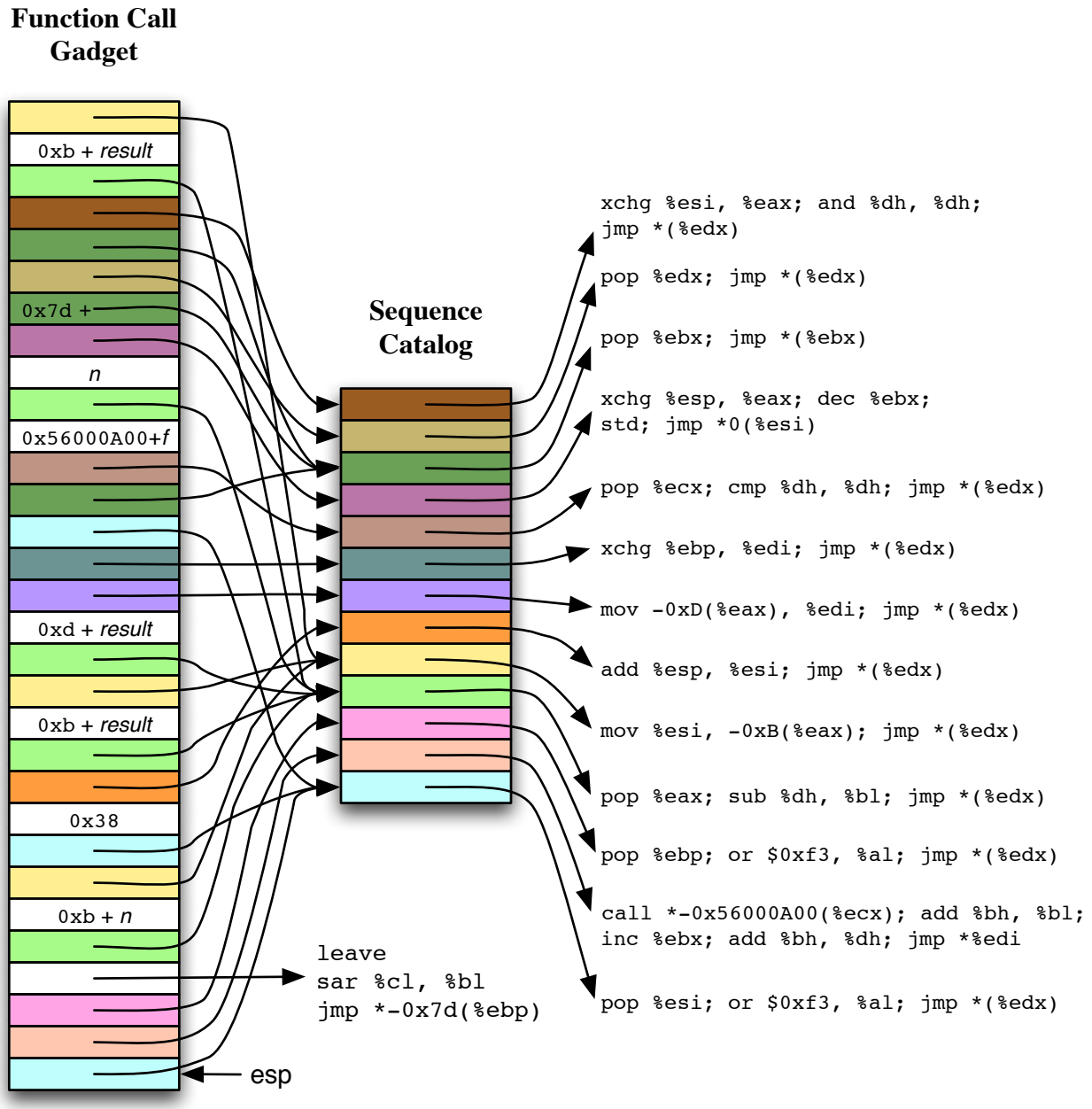

Figure 4: Function call gadget. This convoluted gadget makes the function call $result \leftarrow f(arg_1, arg_2, \ldots, arg_k)$ where the arguments have already been placed at $n+4, n+8, \ldots, n+4k$. The return value is stored into memory at address *result*.

Next, we load esi with the address of the sequence catalog entry for pop *x*; jmp ∗*x*, ecx with the address where the pointer to the function is stored (plus an offset), and eax with the value *n*. Registers esp and eax are exchanged causing the stack pointer to be set to *n*.

Recall that the first thing the function call gadget did was to store the address of the catalog entry for the call-jump sequence to *n*. At this point, the indirect call of the function *fun* happens. After *fun* returns, we cannot rely on the values in registers ecx or edx while eax holds the return value. However, edi holds the address of the leave-jump sequence, thus the jmp ∗%edi instruction causes a leave instruction to be executed which sets the stack pointer to ebp — which is still holding the address we placed into it with the first xchg instruction — and then pops the value off of the top of the stack into ebp. This causes the address of the sequence catalog entry for pop *x*; jmp ∗*x* (plus an offset) to be loaded into ebp causing the subsequent jmp ∗-0x7d(%ebp) instruction to chain the next instruction sequence.

At this point, we have two choices for the implementation. If we do not have a pop %edx; jmp ∗(%edx) sequence, then we can use a popad; jmp ∗(%edx) and lose the return value. In this case, the function call gadget is complete. However, if we do have a pop %edx; jmp ∗(%edx) sequence, then we execute that and then store the return value in eax into memory. This is the form of the gadget shown in Figure 4.

5 Getting Started

Return-oriented programming is an alternative to code injection when an attacker has diverted a target program's control flow by taking advantage of a memory error such as a buffer overflow. How the initial control flow diversion is accomplished, then, is orthogonal to the question of return-oriented programming.[9]

All the same, some of the traditional means of diverting control flow require the target program to execute a return instruction, which means they risk detection by the defenses our new return-oriented programming are designed to evade.

In some cases, a different approach will allow attackers to avoid this initial return. In this section, we discuss four classes of memory errors from the perspective of the pop *x*; jmp ∗*x* return-oriented programming paradigm and consider for each the prospects for an attacker to take control without using a return instruction. Recall that, in order for a return-oriented exploit to be successful, the attacker must gain control of both the instruction pointer and the stack pointer. In addition, the return-oriented program must be some place in memory.

Stack buffer overflow. The traditional means of exploiting a stack buffer overflow is to overwrite the saved instruction pointer in some function's stack frame. When that function returns, control will flow not to the instruction after the call that invoked the function but rather to any location of the attacker's choosing. In a return-oriented attack, this will be the first instruction sequence in the first gadget laid out on the stack; conveniently, the stack pointer will point to the next word on the stack, which is also under attacker control. By this point, however, the LIFO invariant of the return-address stack has been violated. (A single return instruction would not, of course, be caught by defenses that look for several returns in close succession.)

To take advantage of a stack buffer overflow without a return, an attacker must overwrite stack frames while avoiding changing the value of any saved instruction pointers. What she should change is pointer data such as function pointers in a function frame above the one that contains the overflowed buffer. Once the function that contains the buffer has returned (to the function that legitimately called it), the memory around the stack pointer will be controlled by the attacker; when the pointer she modified is used, an instruction sequence such as popad; jmp ∗*y* as its target will allow her to take control of the registers and begin running return-oriented code.

[9] Also orthogonal are defenses against buffer overflows such as stack cookies or generally against reliable exploitation such as address-space randomization.

Setjmp buffer overwrite. The setjmp and longjmp functions allow for nonlocal gotos. A program will allocate space for a jmp_buf structure which consists of at least an array of words long enough to hold registers ebx, edi, esi, ebp, esp, and eip — the callee saved registers. When setjmp is called, it stores the values of those registers into the jmp_buf. The instruction pointer stored into the buffer is the saved instruction pointer pushed onto the stack by the call instruction and the stored stack pointer is the value the esp had before the call to setjmp. When setjmp returns, it returns the value zero in eax.

At some point later, longjmp is called. This restores the general-purpose registers to their previous values, sets eax to the second argument of longjmp, sets the stack pointer, and finally does an indirect jump to the saved instruction pointer. In essence, setjmp returns two times while longjmp never returns.

If an attacker is able to write the exploit program to some location in memory and overwrite two words of a jmp_buf — esp and eip — that is subsequently the first argument to a longjmp call, then the attacker can arrange for his return-oriented exploit to run. This method of transferring control to a return-oriented program is so convenient that it was employed for testing the gadgets described in Section 4. See Section 6 for an example this method.

In the interest of security, GNU libc's setjmp stores the two pointers in the jmp_buf mangled. It first xors the pointers with a fixed value and then rotates the results left 9 bits.[10] In longjmp, the pointers are rotated right and then xored before being used.

C++ vtable pointer overwrite. If the attacker overwrites an object instance of a class with virtual functions on the heap, then there is (in the general case) no hope of controlling memory around the stack pointer. However, the attacker will control the memory around the object itself, as well as around the object's vtable, since in overwriting the object she can cause the vtable pointer to point at some memory under her control, such as a packet buffer on the heap. Depending on the code that the compiler generates for virtual method invocation, then, at the time that an instruction sequence is invoked, one or more registers will point to the object, the vtable, or both. The attacker must leverage these pointers (1) to change the stack pointer to memory she controls, and (2) to cause a second instruction sequence to execute after the first.

Being able to leverage a vtable pointer overwrite to take control in a generic way (i.e., one that depends only on the compiler version and flags, and not on the program being attacked) is at present an open problem. The alternative is to generate an exploit that is specific to the program attacked, the way that, for example, alphanumeric shellcodes must be written differently depending on what register or memory location they can consult to find the shellcode's location [25].

Function pointer overwrite. With a function pointer overwrite on the heap, as with a vtable pointer overwrite, the challenge for the attacker is two fold. The first code sequence she causes to execute must relocate the stack to memory she controls. In the same code sequence, she must arrange for a second instruction sequence to execute in turn. It is likely the case that no generic exploitation technique exists that avoids the use of a return instruction, and a specific exploit must be crafted for each target program.

6 Example Exploit

We construct a complete, working shellcode using a return-oriented program without returns and which contains no zero bytes making it usable with a strcpy vulnerability. Once control flow has transferred to the shellcode, it sets up the arguments for a call to the syscall function:

[10] In a blog post, Ulrich Drepper writes that the value xored is supposed to be a process-specific random value and that he added this pointer "encryption" to jmp_buf, among other places in libc, in December 2005 [8]. On a stock Debian GNU/Linux 5.0.4 ("Lenny") system, this value appears to be constant. Indeed, from a cursory inspection of the source code for GNU libc 2.7 used in this version of Debian, it appears that the random value is supposed to come from the high-precision timer, but that this code is never enabled.

Listing 1: Target program for our example exploit.

```
struct foo
{
        char buffer[160];
        jmp_buf jb;
};

int main( int argc, char **argv )
{
        struct foo *f = malloc( sizeof *f );
        if( setjmp(f->jb) )
                return 0;
        strcpy( f->buffer, argv[1] );
        longjmp( f->jb, 1 );
}


syscall( SYS_execve, "/bin/sh", argv, envp ).
```

The target program, given in Listing 1, allocates enough memory on the heap to hold a 160 byte character array and a jmp_buf. Then, setjmp is called to initialize the jmp_buf and the target program's first argument is copied to the character array. Finally, longjmp causes control flow back to the point of the setjmp's return and the program exits. The target program is compiled and linked with Mozilla's libxul to provide the two instruction sequences pop %ebx; jmp ∗(%ebx) and pop %edx; jmp ∗(%edx) as described in Section 4.

The shellcode "egg" in Listing 2 consists of four parts: (1) the return-oriented program; (2) data used by the program; (3) the instruction sequence catalog; and (4) the data to overwrite the jmp_buf. The program consists of a sequence of pointers to the sequence catalog and values to load into registers. The jmp_buf pointers are overwritten to point the stack pointer at the beginning of the program and the instruction pointer at the instruction sequence pop %edx; jmp ∗(%edx) in libxul. Then, it xors esi with itself to clear it and uses this register to write zero words in the data section as needed. After the zeros have been written, important, nonzero data that was overwritten is restored. Finally, the program ends with a call to the syscall function followed by its arguments which reside in the data.

The pop %edx; jmp ∗(%edx) sequence used is not strictly necessary, it could have been replaced by popad; cld; ljmp ∗(%edx) sequence from libc. This sequence requires the use of a far pointer which contains `00` as its final byte. Normally, strcpy vulnerabilities do not allow zero bytes; however, as part of the copy, a final `00` is written to terminate the string. Thus, our shellcode egg can contain exactly one far pointer at the very end.

When the target program is run with the exploit egg as its first argument, the result is a new shell.

```
steve@vdebian:~/noret/exploit$ ./target "`cat egg`"
sh-3.2$
```

7 Conclusions and Open Problems

We have shown that on the x86 it is possible to mount a return-oriented programming attack without using any return instructions. In the new attack, certain return-like instruction sequences take the place of return instructions. Incidental features of the x86 ISA mean that these sequences are sufficiently frequent to make

Listing 2: Shellcode egg. Each group of four bytes is a single (little-endian) word that makes up the basic unit of return-oriented code and data.

```
0000000: 4cbf0408 40bf0408 34bf0408 14bf0408 3cbf0408  L...@...4.......<...
0000014: 34bf0408 32bf0408 3cbf0408 34bf0408 3bbf0408  4...2...<...4...;...
0000028: 3cbf0408 34bf0408 24bf0408 3cbf0408 38bf0408  <...4...$...<...8...
000003c: 20bf0408 34bf0408 17bf0408 3cbf0408 44bf0408   ...4.......<...D...
0000050: 1cc9045e 48bf0408 0b010101 20bf0408 2cbf0408  ...^H....... ...,...
0000064: 30bf0408 55555555 01273fb7 2f62696e 2f736801  0...UUUU.'?./bin/sh.
0000078: 55555555 20bf0408 01010101 393845b7 f93045b7  UUUU ........98E..0E.
000008c: a97d45b7 ca8a45b7 b98d45b7 115744b7 6779deb7  .}E...E...E..WD.gy..
00000a0: 55aa55aa 55aa55aa 55aa55aa 55aa55aa ee617d1d  U.U.U.U.U.U.U.U..a}.
00000b4: 9122a1ae                                      ."..
```

constructing a Turing-complete gadget set without return instructions feasible given large Linux libraries such as Mozilla's libxul, or libphp.

Because it does not make use of return instructions, our new attack has negative implications for two recently proposed classes of defense against return oriented programming: those that detect the too-frequent use of returns in the instruction stream, and those that detect violations of the LIFO invariant normally maintained for the return-address stack. It does not appear that defenses that maintain a shadow return-address stack can be salvaged. On the other hand, defenses that look for too-frequent use of returns in a program's instruction stream could be modified to look also for too-frequent use of indirect jumps, though this risks a cat-and-mouse game if attackers can switch again to different ways of chaining code sequences.

The major open problem suggested by our work is whether it is possible to find some property that *all* return-oriented attacks provably must share. The use of return instructions to chain sequences appeared to be such a property, but we have shown that it is not. Such a property could be used as part of a defense against return-oriented programming, assuming that it can be efficiently tested. (Indeed, it is not clear that effective defenses against return-oriented programming can be deployed at lower overhead than full control-flow integrity [1, 9].) A second open problem is whether return-oriented programming without returns is feasible on architectures other than the x86.

Acknowledgements

We thank Ahmad Sadeghi, Stefan Savage, and Geoff Voelker for helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 0831532. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *Proceedings of CCS 2005*, pages 340–53. ACM Press, November 2005.

[2] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[3] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC

Advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[4] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In Atul Prakash and Indranil Sengupta, editors, *Proceedings of ICISS 2009*, volume 5905 of *LNCS*, pages 163–77. Springer-Verlag, December 2009.

[5] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In Klaus Julisch and Christopher Krügel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005*, volume 3548 of *LNCS*, pages 32–50. Springer-Verlag, July 2005.

[6] dark spyrit. Win32 buffer overflows (location, exploitation and prevention). *Phrack Magazine*, 55(15), September 1999. `http://www.phrack.org/archives/55/P55-15`.

[7] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In N. Asokan, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Proceedings of STC 2009*, pages 49–54. ACM Press, November 2009.

[8] Ulrich Drepper. Pointer encryption. Blog post, January 2007. `http://udrepper.livejournal.com/13393.html`. Accessed February 4, 2010.

[9] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George Necula. XFI: Software guards for system address spaces. In Brian Bershad and Jeff Mogul, editors, *Proceedings of OSDI 2006*, pages 75–88. USENIX Association, November 2006.

[10] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 15–26. ACM Press, October 2008.

[11] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In Sven Lachmund and Christian Schaefer, editors, *Proceedings of SecuCode 2009*, pages 19–26. ACM Press, November 2009.

[12] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In Dan Wallach, editor, *Proceedings of Usenix Security 2001*, pages 55–65. USENIX, August 2001.

[13] Ralf Hund. Listing of gadgets constructed on ten evaluation machines. Online: `http://pi1.informatik.uni-mannheim.de/filepool/projects/return-oriented-rootkit/measurements-ro.tgz`, May 2009.

[14] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In Fabian Monrose, editor, *Proceedings of Usenix Security 2009*, pages 383–98. USENIX, August 2009.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 2001.

[16] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2001.

[17] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum, January 2010. Online: `http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf`.

[18] Felix "FX" Lidner. Developments in Cisco IOS forensics. CONFidence 2.0, November 2009. Presentation. Slides: `http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf`.

[19] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, September 2003. Online: `http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf`.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of PLDI 2005*, pages 190–200. ACM Press, June 2005.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of PLDI 2007*, pages 89–100. ACM Press, June 2007.

[22] Ryan Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, UC San Diego, March 2009. Online: `https://cseweb.ucsd.edu/~rroemer/doc/thesis.pdf`.

[23] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. Manuscript, 2009. Online: `https://cseweb.ucsd.edu/~hovav/papers/rbss09.html`.

[24] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[25] Berend-Jan "SkyLined" Wever. ALPHA2: Zero tolerance, Unicode-proof uppercase alphanumeric shellcode encoding. Online: `http://skypher.com/wiki/index.php/ALPHA2`, 2004.

[26] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In Andrew Myers and David Evans, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2009*, pages 79–93. IEEE Computer Society, May 2009.