Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming

Andrei Homescu Michael Stewart Per Larsen Stefan Brunthaler
Michael Franz
Department of Computer Science
University of California, Irvine

Abstract

Return-oriented programming (ROP) has gained a lot of popularity lately, as an attack against currently implemented defenses in modern operating systems. Several kinds of ROP-based attacks and anti-ROP defenses have been proposed in recent years. The original attack technique depends on the existence of a hand-picked set of byte sequences (called *gadgets*) in the program, while subsequent approaches use complex scanners, which perform semantic analysis on the code to locate gadgets. The latter ones are efficient at finding gadgets and building an attack, but incur a significant cost in time.

We propose a ROP attack technique, based on a hand-picked but flexible and Turing-complete set of gadgets. One novelty in this approach is the use of microgadgets, which are gadgets restricted to 2 or 3 bytes in length. Our approach splits gadgets into several classes of varying sizes (from 1 to more than 800). Only a single gadget from each class is required for Turing-completeness. The short length of the gadgets, as well as the large size of the classes, increase the likelihood of finding all required gadgets. We also describe an efficient scanner which locates these gadgets in a given program. We then use this scanner on the `/usr/bin` directories from several Linux distributions, to show that many programs indeed contain a Turing-complete set of microgadgets, which attackers can use to perform arbitrary computations.

1 Introduction

In many cases, application code can be useful not only to its legitimate users, but also to attackers which seek unauthorized access to the application. The intruder can then either steal sensitive application data, or hijack its execution. For the latter purpose, the attacker needs to find a way to force the application to execute some code chosen by the attacker. Once this happens, the program can be used for arbitrary purposes, such as attacking other applications or systems.

One important attack technique, so-called *return-into-libc* [16], uses library functions to compromise the system. To get access to those library functions, the attacker only has to manipulate the stack, so that the proper return address and parameters are in the desired positions. Later, Shacham [15] describes a technique called *return-oriented programming*, or ROP for short, which uses existing snippets of code, called *gadgets*, to execute arbitrary algorithms. These gadgets, much smaller than a function, are used as instructions of an abstract virtual machine, which are proven Turing-complete. The attacker chains gadgets together into a translated version of the original attack.

Return-oriented programming is an active research topic, of interest to both the academic community and the security community at large. The original paper on ROP presents a hand-picked set of gadgets, specific to a particular version of `libc`. While it is possible to update the set of gadgets for newer versions, the paper implies that this would be done manually by visual inspection and suggests analyzing other libraries as future work. Follow-up research [14, 7, 17, 6] describes automated systems that scan a binary and build a list of gadgets programatically; in some cases they even compile the attack payload using those gadgets. Some of these tools use various semantic analysis techniques on the gadgets, determining the effects of each gadget before using it.

The evolution of these tools allowed more complex gadgets to be used in an attack, therefore making more binaries vulnerable to ROP-based attacks. At first, scanning was limited to single-instruction gadgets [7], but later research lifted that restriction. Gadgets used for attacks have become more and more complex, with one or more of the following traits:

- Several instructions operating on different registers, or in some cases the same register. This compli-

cates gadget analysis and usage, as the tools must take into account interactions between different instructions.

- Memory operands with arbitrary offsets, which the gadget tool must compensate for using other gadgets.
- Extra instructions in a gadget destroying values in registers. Gadget compilers must use extra gadgets to preserve these values, using *register spilling* techniques.

As opposed to the growing complexity of gadgets used in ROP attacks, we propose a very simple set of hand-picked gadgets with the same computational power: Turing-completeness. We group the gadgets in the set into *classes*. Each class contains a set of microgadgets with equivalent functionality, so that any single microgadget from that class is sufficient to implement the functionality of that class. Some microgadgets from the same class differ just by their input operands, e.g., they operate on different registers, while others perform the same operation in different ways. For example, one way to clear the carry flag on x86 is to use the `CLC` instruction, another is to use `SAHF`. Either of these two options achieves the goal of finding microgadgets to clear the flag.

We designed this set with 3 goals:

Simplicity so we can easily describe and analyze the set of gadgets, without needing to account for any complications or corner cases that other gadget sets present. Our set does not contain gadgets with memory operand offsets, immediate values or more than one instruction.

Ubiquity (or near-ubiquity) so that the set occurs so frequently in the binaries that the threat level warrants attention.

Computational power so the attack can be used successfully for arbitrary code execution or information disclosure.

We picked Turing-completeness as the measurement of computational power, since it is the most powerful target, while also being very clearly specified. While some existing attacks [19] do not rely on this (instead using ROP just as a prelude to x86 code injection), we expect that analyzing a Turing-complete gadget set will provide valuable information about less powerful sets.

To achieve ubiquity, our set must be found in all binaries. While that is not truly possible due to the low frequency of `RET` instructions, we designed our set to achieve a good balance between all three goals. For this reason, the key insight in our choice of gadgets is that *smaller gadgets have a higher probability of occurrence*. Therefore, we attempt to build an attack from gadgets that are as small as possible, while still providing simplicity and power. The smallest useful gadget on the x86 architecture is 2 bytes long: one byte for the proper instruction and another for the return instruction (the `C3` byte). However, we raised the limit on gadget length to 3 bytes, so that we could include more useful operations, like addition and memory accesses.

This paper makes the following contributions:

- In § 2, we present a Turing-complete ROP-based gadget set focusing on short gadgets, so-called *microgadgets*.
- In § 2.5, we describe the applicability of our approach to attacks which use ROP to allocate an executable area of memory, then use that area to copy and execute a second payload.
- In § 3, we analyze the likelihood of success using this attack with a case study on several Linux distributions and several individual binaries from those distributions, showing the success rate of our approach and performance of our scanner.
- In § 3.4, we present a concrete ROP-based exploit for PHP 5.3.2, using microgadgets to load and execute another x86 payload.

2 Description of Gadgets

Our gadget set implements a simple two-address RISC-like instruction set on top of 32-bit x86 microgadgets, using CPU registers as operands. It supports only simple addition, subtraction and logical operations, as well as loads and stores to and from any memory location. We allow all general-purpose integer registers except for `ESP` as microgadget operands. Since each microgadget contains one x86 instruction, other than the `RET`, we restrict the inputs to 2 different input registers and the output to one of the input registers.

For every operation in our set, we hand-picked a list of candidate microgadgets that execute that operation. We derived this hand-picked set from practical experience with code generated by compilers such as `gcc` and `Clang/LLVM`. Looking at larger binaries, we observe the following facts:

- Many useful 2-byte gadgets are very likely to appear, such as `POP reg` and `LAHF`.
- Useful 3-byte gadgets appear in some forms, but not all; for example, an `XOR EAX, EBX` might not always appear, but a `XOR r1, r2` gadget has a high probability of occurence.

Instruction	Contents		
	EAX	EBX	EDX
XCHG EBX, EAX	v_1	v_2	v_3
XCHG EAX, EDX	v_2	v_1	v_3
XCHG EBX, EAX	v_3	v_1	v_2
	v_1	v_3	v_2

Table 1: Exchanging EBX with EDX, using EAX as a temporary. The rightmost columns show the contents of each register before each instruction.

A property of the x86 instruction set is that XCHG EAX, reg is a 1-byte instruction. Therefore, we can implement any exchange or move operation between two registers using combinations of these microgadgets and the EAX register as a temporary. The XCHG reg1, reg2 simply exchanges (or switches) the values of two registers. This approach guarantees that, unlike a set where values are moved around using MOVs, values are not destroyed by such a move. We call this principle *no value left behind*. Table 1 shows an exchange between the EBX and EDX registers.

We rely on a significant number of 2-byte gadgets:

- XCHG EAX, reg to move or exchanging register values. This instruction switches the values of EAX with the value of another register.
- POP reg to load constant values from the stack into the specified register; only one of these is necessary, since we can combine it with XCHG to obtain all the other versions.
- LAHF or PUSHF to read the carry flag, necessary to implement conditional branches.
- XCHG EAX, ESP or LEAVE or POP ESP to copy another register into ESP, also necessary for conditional branches.
- INC reg or DEC reg for various operations described later.
- CLC, SAHF, AAA, DAA, AAS or DAS to clear the carry flag before comparison operations.
- PUSHA to copy ESP into another register in the prologue part of the program .
- LODSD to load a value from the address in ESI into EAX.
- STOSD to store a value to memory.

Using only these instructions, we can already implement several operations: loading a constant into a register, moving a value into another register, incrementing/decrementing a value and, optionally, accessing

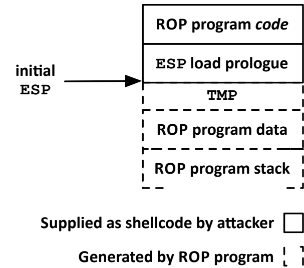

Figure 1: Layout of the ROP program on the x86 stack. The load prologue and ROP program code are supplied by the attacker as shellcode, while the data and stack cells are written by the ROP program itself.

memory. We were unable to obtain Turing-completeness from this limited set, so we will use a set of 3-byte gadgets to offer the following extra operations, which prove sufficient for achieving Turing-completeness:

- Add and subtract two values.
- Perform logical AND, OR, XOR and NOT operations on two values; we only need one of AND/OR and one of XOR/NOT/NEG, since we can emulate one from each pair using the other one and De Morgan's laws.
- Compare two values and branch on comparison result.

These instructions form a Turing-complete language, since our set contains all instructions required to implement an OISC, or *one-instruction set computer* [11]. Our model supports the subneg flavor of OISC, which only requires three operations we support: subtraction, less-than comparison and conditional branching.

2.1 Memory Layout

We use the machine call stack to store both the program's "code" and its data. The preliminary stack-based exploit loads the code on the stack at the address pointed to by ESP. The space on the stack below the code is used as storage for data and temporaries. We reserve the first word just before the code as the temporary variable TMP, which we use as scratch space for various operations. Figure 1 shows the complete layout on the x86 stack.

To access this storage throughout a ROP attack, we need a pointer to some fixed offset on the stack. Therefore, one of the first gadgets in the program has to be some instruction that saves the contents of the ESP register. We preserve this value across the entire execution of the program, moving it through other registers as needed. All stack-relative accesses use the initial ESP value as the base address.

We cannot use an `XCHG` microgadget for this, as we require a non-destructive copy that produces a duplicate, not a move that replaces the original value. We have several other options available:

- `MOV reg, ESP`.
- `ADD/ADC reg, ESP` can be used if `reg` is initialized to 0 and `CF` to 0 for `ADC` (`ADC` is a variant of `ADD` which also adds the carry flag to the result).
- `XOR/AND/OR reg, ESP` can be used if `reg` is initialized to either 0 or $0xffffffff$.
- `PUSHA` pushes all registers on the stack; with an appropriate set of `POP` gadgets executed after it, this microgadget will successfully copy `ESP` into another register.
- `PUSH ESP ; PUSH reg` is a faster 2-instruction variant of `PUSHA`.

Our gadget set requires at least one of these gadgets. The first operations of the program initialize `reg` if needed, followed by the actual copy microgadget.

2.2 Addition, Logical and Memory Operations

Since x86 stores negative values using two's-complement representation, we can reduce subtraction to addition or vice-versa with some simple operations: $a-b=a+(-b)=a+(\bar{b}+1)$. Therefore, we only need to find a gadget implementing one of the four addition/subtraction operations plus negation. We can also use `XOR` for negation, since $\bar{x}=x\oplus 0xffffffff$.

`ADD` and `SUB` are equivalent to `ADC` and `SBB` once we clear the carry flag. To do this, we need a gadget that clears this flag. We previously enumerated several 2-byte instructions that do exactly this. There are also several 3-byte instructions that clear `CF`, when used with the proper register arguments: `ADD`, `ADC`, `SUB`, `SBB`, `AND`, `OR`, `XOR`, `CMP` and `TEST`.

We also need one operation with a memory source, and one operation with a memory destination. To minimize the number of classes, we should design the classes so that as many gadgets are shared between classes as possible. If `LODSD` and `STOSD` for memory loads and stores, we can look for memory access gadgets in the other classes, like the addition and `XOR` classes. The complete list of operations we need is:

- One of `ADD`, `ADC`, `SUB` or `SBB`.
- One `XOR`, `NOT` or `NEG`.
- One of `AND` or `OR`.

```
pop 0 into reg2 here
ADD reg2, reg1
```

Figure 2: Example of register-to-register copy using 0 as the identity value.

- One memory-to-register operation (load), if `LODSD` is not available.
- One register-to-memory operation (store), if `STOSD` is not available.

We combine these to form all addition/subtraction, logical and memory operations.

Since all of these operations have identity values (all ones for `AND`, 0 for the rest), the classes also provide a non-destructive register-to-register copy from `reg1` to `reg2`. It does this by initializing `reg2` with the identity value for an operation, then applying the operation to the pair `reg2, reg1`. Figure 2 shows the steps of this process. We use the same technique for a memory-to-register load, initializing the destination register to the identity value and then using an available memory-to-register operation to load the value. We can also use any `MOV` operations for the last two classes, if available.

The classes handle stores in one of several ways, depending on gadget availability. If the binary contains a direct memory store, we can simply use that. This is an instruction of the form: `MOV DWORD PTR [reg2], reg1`. Otherwise, we have to use a load operation to load the previous value of the location we want to write to, then compute the difference between the old and new value. We apply this difference to the location of the old value. For example, if we only have an addition operation with a memory destination, we have to add the difference between the old value and the new value. This requires that the operation we use also has an inverse, e.g., subtraction is the inverse of addition. An alternative approach is to zero the contents of the destination location (change its contents to 0), then apply our memory store operation afterwards. There are several cases we have to consider:

XOR store We have to use an operation of the form `XOR DWORD PTR [reg2], reg1`. We simply use any available load to get the old value of the cell into a register, then `XOR` that register into the cell. The new contents of the cell will be 0. Then we `XOR` the new value into the cell, getting exactly the effect we were looking for. Figure 3 shows the sequence of operations for this case.

SUB/SBB store We only use this instruction if neither a `MOV`, nor an `XOR` is available. However, some kind of negation operation (`XOR`, `NOT` or `NEG`) with a

```
load [addr_reg] into tmp_reg
XOR DWORD PTR [addr_reg], tmp_reg
XOR DWORD PTR [addr_reg], new_value_reg
```

Figure 3: Sequence of operations to perform a memory store using XOR.

```
load [addr_reg] into tmp_reg
SUB DWORD PTR [addr_reg], tmp_reg
pop 0xffffffff into XOR destination, if using XOR
NEG/NOT/XOR new_value_reg
INC new_value_reg if used NOT
SUB DWORD PTR [addr_reg], new_value_reg
```

Figure 4: Sequences of operations to perform a memory store using SUB or SBB.

register destination is necessary. A store using this operation proceeds in two steps: first, we subtract the old value of the cell from itself, so that the new intermediate value of the cell becomes 0. We do this by loading the value of the cell into a register, then subtracting that register from the destination cell. The second step adds the new value to the cell. Since addition is equivalent to subtracting the two's complement, as shown earlier, we first negate or complement and increment the new value, then subtract it. Figure 4 shows this case. If the only negation we have is a XOR with a memory source, the $0xffffffff$ value has to be accessed from a memory location. We place this value on the stack, then compute its address during execution and use that address in the XOR. Since compilers frequently use FF for padding, we can also use this byte, if available four times consecutively. If no XOR is available, then the NOT/NEG operation we use needs a register operand.

ADD/ADC store This is mostly similar to the previous case, except the old value needs to be negated, not the new one. Figure 5 shows the base case, where a NEG is used.

Once we have a memory load and memory store operation, we can implement all other operations using available classes. When we find a load and a store, it does not matter if each individual instruction has memory, or register operands. We use the TMP scratch space mentioned earlier as a temporary store for memory-load or memory-store instructions. For example, if we only have an ADD with a memory destination, we implement addition and subtraction as stores to TMP followed by loads from that location. The attacker can implement multiplication, division and more complex operations using the operations from this section and conditional branches shown below.

```
load [addr_reg] into tmp_reg
NEG tmp_reg
ADD DWORD PTR [addr_reg], tmp_reg
ADD DWORD PTR [addr_reg], new_value_reg
```

Figure 5: Sequence of operations to perform a memory store using ADD or ADC. An XOR or NOT together with an INC/DEC can replace the NEG, if available.

2.3 Conditional Branches

Our approach only supports conditinal branching on the *less-than* condition directly. This requires only one extra microgadget class, which we describe below. Other equality and ordering comparisons can be implemented indirectly using this single operation.

Since all addition/subtraction operations set CPU flags, we use those to compare two values. Because the carry flag is the least-significant bit of the FLAGS register, it is the easiest to access and use. If we compute the unsigned difference $a-b$, CF is 1 iff $a<b$; conversely, if $a \geq b$ then CF is 0. This inequality comparison can be turned into an equality comparison using the following observation: $a=b$ iff $b \geq a$ and $a \geq b$. The comparisons can also be inverted using negation: $a>b$ iff $-a<-b$. Checkoway et al. [4] use this approach to implement conditional branches in their implementation of ROP without returns.

We have several choices for extracting the carry flag:

- LAHF copies the lowest eight bits of FLAGS into AH. To move the carry flag into the lowest bit, we use unaligned memory operations, which the x86 architecture allows. The program saves EAX in the TMP location using the address stored in a register, then increments that register and loads back the value. After masking out the extra most-significant byte, this becomes equivalent to right-shifting the 3 most significant bytes of the old value. Another masking operation clears out all bits other than CF.

- PUSHF pushed FLAGS on the stack. Unfortunately, using a 2-byte PUSHF microgadget for this purpose is incorrect, as the return instruction would incorrectly jump to the value of FLAGS. However, the extended PUSHF ; PUSH reg microgadgets provide the desired behavior. Figure 6 shows the effects of this microgadget on the stack. The gadget overwrites the preceding stack locations, so additional code is needed that retores those locations to their original values. The terminating RET loads the address of the next gadget from the stack location where reg is saved. Therefore, the value of reg determines the next executed gadget.

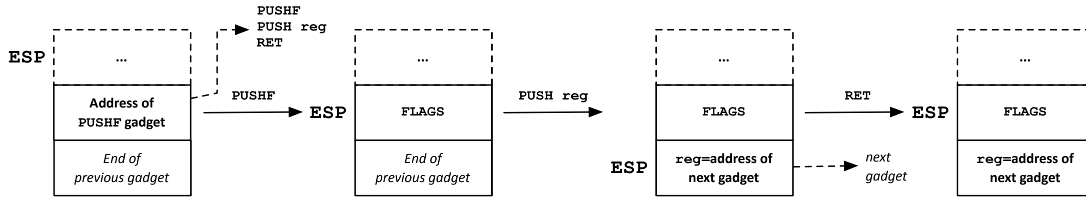

Figure 6: Execution of the `PUSHF ; PUSH reg` microgadget. The `RET` instruction transfers execution to the microgadget whose address was previously stored in `reg`.

- `ADC` and `SBB` also set a value according to the `CF` bit. We use the identity value (0) again to neutralize the arithmetic effect of these operations. Previous ROP models [4] already used *SBB* successfully used to implement conditional branches.

To actually do a conditional jump, we need a way to change the value of `ESP` according to the value of `CF`. The solution to this problem involves some memory address arithmetic, too. We store the new address for the false branch at some address *addr* and the address for the true branch at *addr + 4*. Then, we copy *addr* into a temporary register or `TMP` and add the value of `CF` four times to that register, so that the value in the register becomes `addr + 4 * CF`. Figure 7 shows one possible layout of the addresses and code on the machine stack, when using this technique. We load the contents of the memory word at this address and store it into `ESP`, using one of:

- `XCHG EAX, ESP`.
- `LEAVE` copies `EBP` into `ESP`, so we set the former to the branch target, then use this microgadget.
- `POP ESP` followed by a 32-bit word which is programatically loaded with the branch target. The ROP program has to store the branch target address on the stack once per branch.

We cannot use the `SETC` or the `CMOVC` instructions, which copy the value of `CF` to another register, due to limiting the set to microgadgets of length 3.

2.4 Function and System Calls

The ultimate purpose of the attacker is, in most cases, to take control of the program or the entire system which runs the program. In the latter case, the final stage of the attack entails a call into the operating system (in other words, a *syscall*). There are two ways an attacker can do this: *directly*, through the special instructions provided by the CPU, or *indirectly*, through a library function.

The direct way to do a system call on a Linux distribution on the x86 architecture uses one of two special instructions: `INT 0x80` and `SYSENTER`, both available as two-byte instructions. Since both of these are 2-byte instructions, we can easily extend our set with an extra class of 3-byte gadgets, dedicated to syscalls. Using either of these instructions, the attacker can transfer control to the operating system. Before executing the instruction, the attacker has to put the arguments of the call into the proper registers. Since our gadget set provides access to all general-purpose registers except for `ESP`, the exploit code can easily place all the proper values into the correct registers.

The indirect way to call into the operating system uses a library function, most likely from `libc`. On 32-bit x86 Linux system, the ABI (*Application Binary Interface*) specifies that all parameters are placed on the stack before a function call. Therefore, the exploit code only needs to place the function arguments on the stack at the current `ESP`, then transfer control to the chosen function. Just before the call, the attacker sets `ESP` to the address of the first argument; we already provide a gadget class that sets `ESP`.

Figure 8 shows the memory layout of the code and data for this operation. Before the ROP program can call into the library function, it must reserve a stack zone for the arguments and function address. Then, it copies the values of the arguments to that zone, preceded by the address of the function and the pointer to a return gadget. The last gadget in this sequence stores the base address of that zone into `ESP`; since gadgets always end in a `RET`, that return instruction loads the function address from the new `ESP` and jumps to that address.

The stack zone starts with the address of the function, so that the call gadget transfers control using a `RET`. After that transfer, the stack pointer moves to the next position, where the *function return address* is stored. When the function ends, it executed its own `RET` instruction which loads the next gadgets from the `return address` slot. We point this address at a gadget meant to reset the stack pointer to the next gadget to be executed by our ROP program. We save this return stack

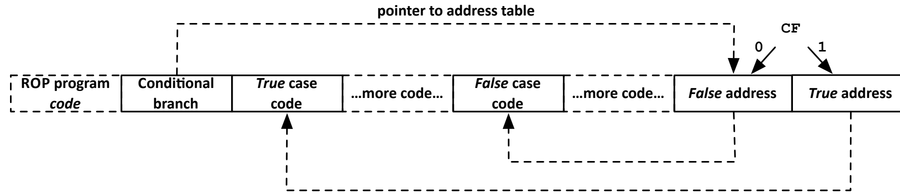

Figure 7: ROP code and address table layout for conditional jumps. The 2-element table contains 2 stack pointers, one for the `true` case and one for the `false` case. The jump code reads the appropriate value from the table, then sets `ESP` to that value.

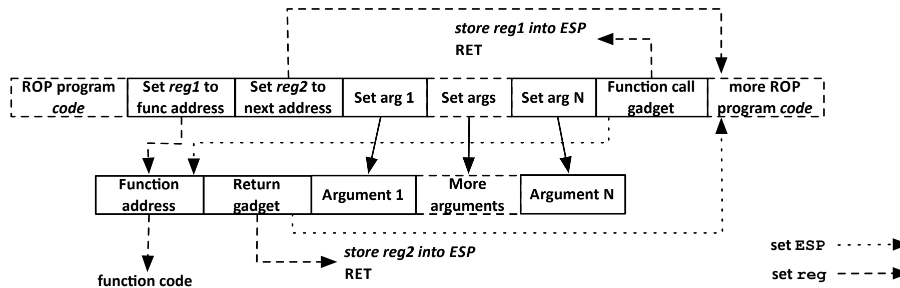

Figure 8: ROP code and data layout for a function call.

position in a register `reg2`, which we must set before the call and copy to `ESP` just after. In order to prevent the function from destroying the contents of this register, we must pick a callee-saved one: `EBX`, `EBP`, `ESI` or `EDI`.

2.5 A Practical Subset of Classes

While Turing-completeness is a useful and interesting goal in the design of ROP attacks, it often offers more operations than the attacker needs. In practice, ROP has been often used as the early stage of an attack, where a short ROP payload is used to execute a longer x86 payload. The ROP code first calls an OS function to allocate a writable and executable zone, usually `mmap` on Linux or `VirtualAlloc` on Windows. The code then copies the full x86 payload to that zone, then transfers control to the zone using an indirect branch instruction. Any set of ROP gadgets that supports these operations is practically useful.

Since the ROP payload needs to do very little computation in such an attack, many of the operations supported by the earlier set become superfluous. For example, the set does not need to support loops, since the only loop in the payload simply copies the x86 payload to memory, and that loop is easily unrolled. The only critical operations that a restricted set of classes for these ROP trampolines are:

- Storing a constant value to a memory location, so the program can copy the x86 payload to the address returned by the allocation function.
- Loading a value from a memory address or branching to an address stored in memory (the *indirect-branch-to-memory* operation), to support calling a function from `libc`. This is needed on Linux because the OS loader stores the address of the allocation function, i.e. `mmap`, in the GOT. The ROP payload either loads the address of `mmap` from memory into a register, then jumps to the contents of that register, or jumps directly to the address loaded from the GOT.
- All gadgets required for function calls, including gadgets required to continue execution of the ROP program after the function returns, e.g., `LEAVE`.

The single function call made by this attack is much easier to set up than a general function call. All function arguments have constant values, so the attacker can simply place them inside the payload. The payload itself contains the entire stack zone described in § 2.4. The only complication is the presence of the arguments on the stack. These arguments do not contain valid gadget addresses. For this reason, the ROP payload uses the function return gadget to move the stack pointer past the arguments. The simplest solution is to use `LEAVE` for

this purpose, after having set EBP to the correct stack location. However, this requires that the attacker predict a correct value for EBP, which is difficult in the presence of ASLR.

Another possible solution uses the POPA instructions, which pops all 8 registers from the stack from 8 consecutive slots. Since mmap takes 6 arguments, POPA would be sufficient to skip over the arguments. However, it cannot be used since one of the 8 registers is EAX, which is the same register that contains the result of mmap. Using POPA would require that EAX be saved to memory beforehand, which cannot be easily combined with POPA in a single microgadget. This approach proved too inflexible to use in our set of classes.

The ASLR-proof solution we chose programatically computes the correct EBP that the program copies to the stack pointer after the function. The goal is to increment ESP at the end of the function by a value picked by the attacker. One way to do this uses ADD ESP, immediate gadgets. However, these gadgets are not very frequent and have strict limits on the choice of the immediate value. A more flexible, yet more complicated solution, is to split this operation into several ones:

- Copy the stack pointer into some register reg.
- Load the constant offset into a register.
- Use register-to-register addition / subtraction / exclusive-or to adjust the stack pointer.
- Copy the result back to EBP, using LEAVE as the return instruction.

Since the addition operations are only available in 3-byte microgadgets, we once again require the full set of registers. However, we observe that the *no value left behind* requirement is considerably weakened; there are only two values that must be preserved across many gadgets: the computed stack pointer and the return value of the function (stored in EAX). We can use gadgets that destroy all other registers, a property which we use to extend the register exchange classes. In addition to XCHG, we can also use MOV and many of the previously presented instructions to perform register-to-register copies. However, we still use EAX as a scratch register and do copies in several steps; therefore, for any register reg, we require one class to copy EAX to reg and one class for the reverse operation.

3 Evaluation

3.1 Microgadget Classes

The first step of our evaluation consists of building the concrete sets of gadgets, based on the classes we describe in § 2. We automatically identify valid microgadgets for each class and count the size of each class, shown in Table 2. The more gadgets each class contains, the easier it is to find it in a binary.

#	Class	Size	2-byte	3-byte
1	INC/DEC	14	✓	
2	POP reg	7	✓	
3	ADD/ADC/SUB/SBB	624		✓
4	XOR/NOT/NEG	482		✓
5	AND/OR	312		✓
6	*load from memory*	295	✓	✓
7	*store to memory*	217	✓	✓
8	*clear CF*	839	✓	✓
9	*load flags*	372	✓	✓
10	*load ESP*	106	✓	✓
11	*set ESP*	101	✓	✓
12	XCHG EAX, EBX	1	✓	
13	XCHG EAX, ECX	1	✓	
14	XCHG EAX, EDX	1	✓	
15	XCHG EAX, EBP	1	✓	
16	XCHG EAX, ESI	1	✓	
17	XCHG EAX, EDI	1	✓	

Table 2: The set of microgadget classes and the number of microgadgets in each class. The two rightmost columns show whether each class contains at least one microgadget of that specific size.

Table 2 also shows a subset of classes with one element each: the XCHG EAX, reg classes. These classes are required for Turing-completeness, but each class only contains one gadget. For this reason, we expect these six classes to have a large impact on our results. To get a better idea on the impact the XCHG EAX, reg have on vulnerability rate, we performed two sets of scans: one with the full gadget set and one without the XCHG instructions. We believe that future extensions to the *no value left behind* approach with larger classes and more instructions will increase the flexibility of microgadgets.

Our design for the simpler loader payload described in § 2.5 shows a similar number of classes: 11 classes for the non-ASLR version and 35 classes for the ASLR-proof version. The latter has a larger number of gadgets because there are 2 separate copy classes for each register, and there are 6 registers other than EAX and ESP.

3.2 Threat Evaluation

To measure the vulnerability of actual programs susceptible to our attack, we implemented a scanner in Python and ran it on large sets of binaries. This scanner parses each executable file and determines whether each program contains the complete set of microgadget classes. We selected a sample of Linux distributions released by

different groups and in different years, so we could evaluate across a range of scanned program and compiler versions. Tables 3 and 4 show the list of distributions we scanned, the number of vulnerable binaries we found and the time needed to scan each distribution. We set up a VMWare Fusion virtual machine with 1 GiB of memory and 20 GB of disk space for each Linux system; the host system consisted of an Intel Core i7 system with 8 GiB of memory and an 128 GB SSD and we performed a clean installation of each distribution.

For every distribution, we run our scanner on the executable sections of each file in `/usr/bin`. We use this directory as it usually has the largest concentration of Linux binaries on a system, as most Linux applications get installed there. We ran our scanner in two modes: *independent* mode, or *i-mode*, where we scan each binary on its own, and *complete mode*, or *c-mode*, where we also recursively scan all libraries that each binary depends on. The difference between these two modes is significant, as the binaries themselves are not subject to ASLR, when it hasn't been compiled as *position-independent code*. By default, Linux programs aren't compiled with this options, as it comes with a performance hit [12]. If a binary contains the full set of microgadgets without using any libraries, then that binary can be attacked easily on many different systems, as the programs themselves are not loaded at a random address on Linux. On the other hand, our results show that many more programs are vulnerable when considering their dependencies, particularly since some programs are just small loaders for larger libraries, which implement the actual functionality.

While the exchange subset is required for Turing-completeness, our results show that its small size has a large impact on the frequency of the entire set. Scanning without the exchanges showed an upper bound on the vulnerability rate from our model, absent any future extensions to the 11 large gadget classes. Removing the exchange subset proved to have a significant impact on success rate.

To investigate the impact of binary size on availability of the gadgets, we also split the binaries into groups by file size and scanned them in *i-mode*. Table 5 shows the results of this scan, excluding all files smaller than 3MB. While the sample size is small (there are few binaries in the $>$ 1MB range on Linux), the table shows that almost all the large ones contain all the required gadgets for Turing-completeness.

We also ran our scanner on the two separate versions of the restricted classes, which provide enough functionality to call `mmap` and copy an arbitrary payload to memory. The ASLR-proof version also includes the necessary gadgets to compute the new value of `ESP` after `mmap` returns. Our scans show that the simpler no-ASLR set occurs much more frequently than the Turing-complete set, due to the former's simplicity. However, the ASLR-proof set is complex enough that it occurs almost as rarely as the Turing-complete set. In two particular cases, PCLinuxOS and OpenSUSE, the `mmap`-only set actually occurs in fewer binaries. This happened because some of our simplifications, such as the complete removal of classes supporting logical operations, also eliminated some valid candidates from the surviving classes.

Distribution	no `XCHG` *c-mode*	`XCHG` *c-mode*	no `XCHG` *i-mode*	`XCHG` *i-mode*
CentOS 6.0	3m30s	3m14s	1m36s	1m35s
OpenSUSE 11.4	2m45s	3m15s	1m36s	1m34s
PCLinuxOS '11	4m16s	3m31s	1m56s	1m48s
Fedora 15	3m40s	3m34s	1m26s	1m51s
Kubuntu 7.10	2m00s	2m04s	0m59s	0m57s
Kubuntu 11.10	2m09s	2m06s	1m00s	1m00s
Ubuntu 9.04	1m52s	2m19s	1m30s	1m15s
Ubuntu 10.04	2m04s	2m05s	1m03s	1m01s

Table 4: Time required to scan all of `/usr/bin` on several Linux distributions.

3.3 Case Study: Web Browsers

Since web browsers are large, complex applications that virtually every modern networked system uses, we decided to scan two popular ones: *Firefox* and *Google Chrome*. We used the version of Firefox available by default on each distribution. Firefox is split into 2 separate parts: the launcher binary, *firefox-bin*, which is around 64KiB in size, and the dynamic library (*libxul.so*) which contains a major part of the implementation. We scanned the dynamic library itself in *i-mode*, including all exchanges, separately from our previous `/usr/bin` scan. Table 6 shows the compiler and browser versions installed on the systems we scanned. On almost all Linux and Firefox versions we scanned, this library contained all the needed gadgets for Turing-completeness. We couldn't scan the `libxul.so` file from Firefox 2.0.0.6 because that version of the browser doesn't use this library; the browser code is split into several smaller libraries.

We also scanned Chromium, the open-source version of Google Chrome. We installed and scanned the latest version[1] of these packages provided by Google, in both the `.rpm` and `.deb` versions. Both versions provided the browser as one large binary (around 80MiB in size). The two binaries contained Turing-complete sets of gadgets, even when scanning in *i-mode* with exchanges.

[1] The Chromium version we tested was 15.0.874.106.

Distribution	All	no XCHG *c-mode*	XCHG *c-mode*	no XCHG *i-mode*	XCHG *i-mode*	mmap-only payload no ASLR	mmap-only payload ASLR-proof
	Total binaries						
CentOS 6.0	2231	783	340	20	7	24	10
OpenSUSE 11.4	2292	1804	323	77	18	58	16
PCLinuxOS '11	2405	955	442	56	13	26	11
Fedora 15	1881	758	322	39	15	40	16
Kubuntu 7.10	1337	404	262	27	8	17	9
Kubuntu 11.10	1655	565	271	45	14	39	15
Ubuntu 9.04	1492	434	212	31	4	21	6
Ubuntu 10.04	1587	497	164	35	10	24	12

Table 3: Number of vulnerable binaries from /usr/bin on several Linux distributions. The mmap-only columns show the availability of a set of classes that allow the attacker to execute an arbitrary x86 payload.

Distribution	3-4MB	4-5MB	5-6MB	6-7MB	7-8MB	8-9MB	9-10MB	>10MB
CentOS 6.0	0 / 0 / 7	1 / 2 / 2	0 / 3 / 3	0 / 0 / 2	5 / 7 / 7	0 / 0 / 1	0 / 0 / 0	1 / 1 / 1
OpenSUSE 11.4	1 / 8 / 10	2 / 2 / 4	0 / 0 / 0	0 / 1 / 1	7 / 8 / 8	1 / 1 / 1	0 / 0 / 0	3 / 3 / 3
PCLinuxOS '11	1 / 3 / 5	0 / 1 / 3	0 / 0 / 2	0 / 1 / 1	5 / 7 / 7	1 / 1 / 1	3 / 4 / 4	1 / 1 / 1
Fedora 15	0 / 8 / 22	1 / 1 / 2	1 / 1 / 1	0 / 1 / 1	5 / 7 / 7	2 / 2 / 2	3 / 3 / 3	2 / 2 / 2
Kubuntu 7.10	0 / 3 / 5	0 / 0 / 0	2 / 2 / 2	2 / 3 / 3	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
Kubuntu 11.10	1 / 6 / 8	1 / 2 / 3	0 / 0 / 0	1 / 2 / 2	5 / 7 / 9	2 / 3 / 3	1 / 1 / 1	3 / 3 / 3
Ubuntu 9.04	1 / 4 / 9	0 / 1 / 4	0 / 2 / 2	2 / 3 / 3	1 / 2 / 2	0 / 0 / 0	0 / 1 / 1	0 / 0 / 0
Ubuntu 10.04	1 / 3 / 5	0 / 3 / 3	4 / 5 / 7	3 / 3 / 3	0 / 3 / 4	0 / 0 / 0	2 / 2 / 2	0 / 0 / 0

Table 5: Vulnerable with XCHG / without XCHG / total binaries grouped by file size, with browsers and files smaller than 3MB excluded.

Distribution	Version Firefox	Vulnerable Firefox	Vulnerable Chromium
CentOS 6.0	3.6.9	✓	not installed
OpenSUSE 11.4	4.0b12	✓	✓(.rpm)
PCLinuxOS'11	5.0	✓	✓(.rpm)
Fedora 15	4.0.1	✓	not installed
Kubuntu 7.10	2.0.0.6		not installed
Kubuntu 11.10	9.0.1	✓	✓(.deb)
Ubuntu 9.04	3.0.8	✓	not installed
Ubuntu 10.04	3.6.18	✓	not installed

Table 6: GCC and Firefox versions on each distribution we scanned.

Gadget	File Offset	Memory Address
POP EBP	0x3035a0	0x834b5a0
LEAVE	0x30fdad	0x8357dad
POP EAX	0x233125	0x827b125
JMP *[EAX]	0x109da1	0x8151da1
XCHG EAX, EDI	0x279712	0x82c1712
STOSD	0x038303	0x8080303
DEC EAX	0x1a5567	0x81ed567
PUSH EAX	0x27c206	0x82c4206

Table 7: Locations of gadgets in PHP 5.3.2 binary.

3.4 ROP in Practice

To prove the feasability of an attack using microgadgets, we designed a payload and ran it on a popular network-facing program. PHP is one of the most popular server-side languages, so we used the official PHP interpreter to test an attack. ROP attacks require an attack vector to load the ROP program into the target process and execute the program. For this purpose, we used the vulnerability described in CVE-2011-1938 [13] as a precursor to ROP, targeting PHP version 5.3.2. As an x86 payload, we used a piece of shellcode from the Metasploit [1] framework, which we copied into our attack, with a very small change. We used Ubuntu 10.04 as host operating system.

Modern compilers secure compiled binaries using several techniques, such as stack canaries, meant to detect control flow hijacking. Since the default build of PHP on Ubuntu is compiled with these security features enabled, we built our own version of PHP, with all security features disabled. We then scanned the resulting binary with our scanner and found all required gadgets for both the non-ASLR classes and the ASLR-proof classes. Table 7 shows the addresses of the non-ASLR gadgets in the PHP binary. These 8 microgadgets provide all the required classes. We used these gadgets to build both ROP payload as described in § 2.5 and successfully ran both of them. The program opened a shell using both payloads. Appendices A and B show the sequences of microgadgets used in the attacks.

As mentioned earlier, the x86 payload we used from Metasploit did not work by default. At first, the program

initializes this address to the value returned by `mmap`. The x86 payload was written so that execution starts from the beginning. However, our memory store operations incremented the destination address after each write. Therefore, the payload copy algorithm moved the address to the end of the payload. To compensate for this, we added a backwards jump to the beginning as the last instruction of the payload, then moved the address back two bytes to point to this jump.

4 Related Work

One arbitrary code execution technique that does not rely on code injection is *return-to-libc* [16]. This technique redirects the control flow of the program to one or more functions found in the standard system libraries, allowing the attacker to compromise the target program or even the entire system. Exploits using this technique only set up the stack so that a system call like `system` is possible. Recent research shows that this set of gadgets is Turing-complete when the target binaries use the standard C library [18].

The original *return-to-libc* approach uses entire library functions as the basic elements of the attack code. Krahmer [9] presents a simpler technique, which uses small snippets of code from inside libraries or the program itself. These snippets are called *borrowed code chunks*. The only restriction on the chunks was that each chunk was responsible for transferring execution to the following chunk. Shacham [15] formalizes these attacks (which he calls *return-oriented programming*) and shows that Turing-completeness is possible using just the C library available on a Linux system, and chunks ending in a `RET` instruction, so-called *gadgets*. Follow-up papers extend this work to RISC architectures [3, 5, 8], and to different branching instructions, like `POP/JMP` pairs [10] and jumps in general [2].

The original ROP paper [15] describes a hand-picked set of gadgets. Different compilers or compilation options arrange gadgets in the target binary or libraries differently, or maybe remove some gadgets completely. Shacham uses combinations of complex gadgets to implement operations such as addition and conditional branching, since simpler versions are not available.

Subsequent efforts [7, 6, 17, 14] focus on automated construction of valid and Turing-complete sets of gadgets. These tools take as input a binary and a small program written in a simplified gadget-specific language, scan the binary and then build a list of gadgets in the binary. Next, they compile the input program into a "gadget program", which is the sequence of gadgets which implements the attack code.

The most significant problem in this approach is matching gadgets (and their semantics) to some desired behavior. Previous solutions simply limit the gadgets to a few instructions (the Constructor [7] limits gadgets to one instruction) and use one of those instructions to implement each desired operation. Another implementation [6] determines semantics by reducing each gadget to an expression tree which captures the gadget's behavior. `Q` [14] identifies the semantics of a gadget using a semantic analysis step based on *postconditions*, then uses a similar tree-based approach to match the semantics to the behavior of the attack code. Another approach [17] use a SMT solver to rewrite each gadget as a series of binary functions of the input bits, where each function corresponds to an output bit. This way, the problem reduces to matching boolean functions between the gadgets and the target code.

5 Discussion and Future Work

Our initial approach to *no value left behind* can be improved using extensions to the set of exchanges. We believe designing more complicated sets that satisfy the *no value left behind* requirement can extend our attack to a larger set of binaries. Also, we note the entire gadget set can be extended further, adding more gadgets to the current classes or eliminating some classes completely. For example, we can extend all classes which contain 2-byte microgadgets with 3-byte versions of the same microgadgets, where the middle byte is a `NOP` instruction or similarly harmless operation.

The classes of microgadgets presented here target 32-bit x86 systems, using 32-bit values and memory addresses. We can trivially reuse most classes of gadgets presented here on 64-bit systems, with one exception: the new 64-bit instruction set makes the increment/decrement classes invalid. On 64-bit x86, the 1-byte encodings for `INC`/`DEC` have been reassigned to the new `REX` family of prefixes. In an attack targeted at 64-bit systems, increment/decrements would use the 2-byte versions of these operations, therefore requiring 3-byte microgadgets. One direction for future research would look into using alternatives for the increment/decrement operations.

A second important feature of 64-bit systems are 64-bit addresses. This difference has a significant impact on microgadgets, since some registers store addresses used for memory load, store or indirect jump operations. One such register is `EBP`. On Linux systems, stack addresses use more than 32 bits, so a ROP program accessing the stack cannot use 32 bit registers to store stack addresses. Therefore, memory operations would use 64-bit registers, which require an extra 1-byte `REX` prefix. In addition to memory operations, all other operations with operands or results used as memory addresses need to support 64-bit registers. Due to these extra requirements,

many gadgets in our gadget classes would grow by an extra byte. Investigating the impact of this extra byte on the ubiquity of 64-bit microgadgets is another direction for future research.

6 Conclusions

We presented a Turing-complete set of 2- and 3-byte gadgets, which can be used to perform arbitrary operations and call into the operating systems. We also showed two non-Turing-complete sets that can be used to load an arbitrary x86 payload into memory, then execute it. Our evaluation showed that these gadgets occur with significant frequency (up to a third of all binaries on a Linux system, when considering shared libraries), making it a good step towards full understanding of ROP. In our evaluation, it also took a few minutes to scan thousands of binaries. This lowers the cost to an attacker to locate the needed gadgets, when targeting a specific binary. The attack itself or some preliminary trampoline could contain our microgadget scanner, making a Turing-complete set of gadgets available with relative ease.

A simple, powerful, yet frequent enough set of gadgets is very useful to understand and quantify the threat of return-oriented programming. While previous tools provide information on gadgets by scanning binaries and generating gadget databases, our set provides very useful a priori knowledge of the gadgets an attacker might use. This information could be used to design new defenses, simulate attacks or estimate the likelihood of a successful attack.

Acknowledgments

We thank the anonymous reviewers and our shepherd Sean Heelan for the valuable comments and suggestions.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. D11PC20024. 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 Defense Advanced Research Projects Agency (DARPA) or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate, Sierra Vista Branch.

References

[1] *Metasploit Project*, 2010. http://www.metasploit.com/ (April 2011).

[2] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 30–40.

[3] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), ACM, pp. 27–38.

[4] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), ACM Press, pp. 559–72.

[5] DAVI, L., DMITRIENKO, A., SADEGHI, A., AND WINANDY, M. Return-oriented programming without returns on ARM. Tech. rep., System Security Lab, Ruhr University Bochum, Germany, 2010.

[6] DULLIEN, T., KORNAU, T., AND WEINMANN, R. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–.

[7] HUND, R., HOLZ, T., AND FREILING, F. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium* (Berkeley, CA, USA, 2009), USENIX Association, pp. 383–398.

[8] KORNAU, T. Return-oriented programming for the ARM architecture. Master's thesis, Ruhr University Bochum, Germany, 2009.

[9] KRAHMER, S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques. Online, September 2005. www.suse.de/~krahmer/no-nx.pdf (February 2012).

[10] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating Return-oriented Rootkits with "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 195–208.

[11] MAVADDAT, F., AND PARHAMI, B. *URISC: the Ultimate Reduced Instruction Set Computer*. Research report. Fac. of Mathematics, Univ., 1987.

[12] MOSER, J. Debian SbD: Position independent executables. http://d-sbd.alioth.debian.org/www/?page=pax_pie (February 2012).

[13] PHP socket_ connect() stack buffer overflow, 2011. (CVE-2011-1938).

[14] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium* (2011), USENIX Association.

[15] SHACHAM, H. 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* (2007), ACM Press, pp. 552–561.

[16] SOLAR DESIGNER. *Getting around non-executable stack (and fix)*, 1997. Bugtraq mailing list, http://www.securityfocus.com/archive/1/7480 (April 2011).

[17] SOLE, P. Hanging on a ROPe. In *ekoParty Security Conference* (2010). http://www.immunitysec.com/downloads/DEPLIB20_ekoparty.pdf.

[18] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V. W., AND NING, P. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th Interntional Symposium on Recent Advances in Intrusion Detection* (2011). To appear.

[19] ZOVI, D. Practical ROP, 2010. http://www.trailofbits.com/resources/practical_rop_slides.pdf.

Appendix A: No-ASLR Payload

```
POP EBP
<ESP after function>
POP EAX
<address of mmap in GOT>
JMP *[EAX]
LEAVE
<argument 1: addr=NULL>
<argument 2: length=0x1000>
<argument 3: prot=0x07>
<argument 4: flags=0x22>
<argument 5: fd=-1>
<argument 6: offset=0>
-- Function returns here --
XCHG EAX, EDI
POP EAX
<4 bytes of payload>
STOSD
POP EAX
<4 more bytes of payload>
STOSD
...
POP EAX
<last 4 bytes of payload>
POP EAX
XCHG EAX, EDI
DEC EAX
DEC EAX
PUSH EAX ; RET
```

Appendix B: ASLR-proof Payload

```
-- Load ESP into EBP using PUSHA --
POP EAX
<address of POP EAX>
XCHG EAX, EDI
POP EBP
<address of POP EBP>
POP EBX
<address of POP EAX>
POP EAX
<address of POP EAX>
XCHG EAX, ECX
PUSHA
-- Add delta to EBP --
POP EAX
<delta of ESP to add: 60>
XCHG EAX, EDX
XCHG EAX, EBP
ADD EAX, EDX
XCHG EAX, EBP
-- rest continues as no-ASLR version>
POP EAX
<address of mmap in GOT>
...
```