

The Geometry of Innocent Flesh on the Bone

Hovav Shacham*
Weizmann Institute of Science
hovav.shacham@weizmann.ac.il

April 3, 2007

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. This makes our attack resilient to defenses that strip functions such as `system` from `libc`. Unlike previous attacks, ours combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to build such gadgets using the short sequences we find in a specific distribution of GNU `libc`, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our *thesis*.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing `libc` to recover the instruction sequences that can be used in our attack.
2. Using sequences recovered from a particular version of GNU `libc`, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what we call, facetiously, *return-oriented programming*.
3. In doing the above, we provide strong evidence for our thesis and a template for how one might explore other systems to determine whether they provide further support.

In addition, our paper makes several smaller contributions. We undertake a study of the provenance of `ret` instructions in the version of `libc` we study, and consider whether unintended `rets` could be eliminated by compiler modifications. We show how our attack techniques fit within the larger milieu of return-into-libc techniques, and give a narrative of attacks and defenses that situates recent developments and predicts future ones. We implement and test each of our gadgets.

*Supported by a Koshland Scholars Program fellowship.

1.1 Background: Attacks and Defenses

Consider an attacker who has discovered a vulnerability in some program and wishes to exploit it. Exploitation, in this context, means that he subverts the program's control flow so that it performs actions of his choice with its credentials. The traditional vulnerability in this context is the buffer overflow on the stack [1], though many other classes of vulnerability have been considered, such as buffer overflows on the heap [28, 2, 13], integer overflows [33, 11, 4], and format string vulnerabilities [24, 10]. In each case, the attacker must accomplish two tasks: he must find some way to subvert the program's control flow from its normal course, and he must cause the program to act in the manner of his choosing. In traditional stack-smashing attacks, an attacker completes the first task by overwriting a return address on the stack, so that it points to code of his choosing rather than to the function that made the call. (Though even in this case other techniques can be used, such as frame-pointer overwriting [14].) He completes the second task by injecting code into the process image; the modified return address on the stack points to this code. Because of the behavior of the C-language string routines that are the cause of the vulnerability, the injected code must not contain NUL bytes. Aleph One, in his classic paper, discusses how to write Linux x86 code under this constraint that `execs` a shell (for this reason called "shellcode") [1]; but shellcodes are available for many platforms and achieving many goals (see, e.g., [30]).

This paper concerns itself with evaluating the effectiveness of security measures designed to mitigate the attacker's *second* task above. There are many security measures designed to mitigate against the first task — each aimed at a specific class of attacks such as stack smashing, heap overflows, or format string vulnerabilities — but these are out of scope.

The defenders' first gambit in preventing the attacker's inducing arbitrary behavior in a vulnerable program was to prevent him from executing injected code. The earliest iterations of this defense, notably Solar Designer's StackPatch [26], modified executables' page tables to make the stack nonexecutable. Since in stack-smashing attacks the shellcode was typically injected onto the stack, this was already a useful defense. A more complete implementation, dubbed " $W\oplus X$," ensures that no page in a process image is marked both writable (" W ") and executable (" X "). With $W\oplus X$, there is no location in memory into which the attacker can inject code to execute. The PaX project has developed a patch for Linux implementing $W\oplus X$ [21], and similar protections are included in recent versions of OpenBSD. Intel recently added to its processors a per-page execute disable (" XD ") bit to make $W\oplus X$ implementations more efficient.

Now that the attackers cannot inject code, their response is to use, for their own purposes, code that already exists in the process image they are attacking. (It was Solar Designer who first suggested this approach [27].) Since the standard C library, `libc`, is loaded in nearly every Unix program, and since it contains routines of the sort that are useful for an attacker (e.g., wrappers for system calls), it is `libc` that is the usual target, and such attacks are therefore known as return-into-`libc` attacks. But in principle any available code, either from the program's text segment or from a library it links to, could be used.

By carefully arranging values on the stack, an attacker can cause an arbitrary function to be invoked, with arbitrary arguments. In fact, he can cause a series of functions to be invoked, one after the other [19]. Return-into-`libc` attacks, therefore, can be every bit as devastating as standard code injection attacks — provided that useful functions exist in the process image and that the attacker knows their addresses.

What is to be the defenders' next move? We consider two possibilities: first, they can try to make it difficult for attackers to guess the addresses of `libc` functions; second, they can try to remove from `libc` those functions that an attacker might wish to invoke.

The first option is called address-space layout randomization (ASLR), and is in fact implemented by the PaX project [20] and in OpenBSD. These implementations randomize the base address at which libraries (and, with appropriate toolchain support, the executable's text segment) are loaded as well as the location of

the stack; in a 32-bit address space, they provide approximately 16 bits of entropy for code and 20 bits for the stack. A paper of Shacham et al. [25] shows a class of return-into-libc exploits for which the relevant entropy can be searched by brute force in just a few minutes, even over a network. In addition, the paper argues that it would be difficult to modify ASLR to increase the entropy while keeping the Unix dynamic linking system in place. This “derandomization” attack suggests that, for 32-bit platforms at least, ASLR is an insufficient defense.

What about the second option? At first it might seem that the defenders can make some progress. After all, the function usually called in return-into-libc attacks, `system`, is not much used by Unix programs.¹ Remove it, and the attackers must rewrite their exploits to use some other, less-useful functions. But now what logically ensues is a sort of negativeland arms race, in which defenders remove ever more functions from libc and attackers rewrite their exploits in ever more convoluted ways.

The question is, Who will win?

1.2 Our Results

We answer the question posed immediately above. Our analysis suggests that, on the x86, the attackers will win. It does not matter which (or how many) functions the defenders choose to remove. We demonstrate that a return-into-libc attack can be mounted that calls *no functions at all*.

The building blocks for the traditional return-into-libc attack are functions, and these can be removed by the maintainers of libc. By contrast, the building blocks for our attack are short code sequences, each just two or three instructions long. Some are present in libc as a result of the code-generation choices of the compiler. Others are found in libc despite not having been placed there at all by the compiler. In either case, these code sequences would be very difficult to eliminate without extensive modifications to the compiler.

To understand how there exist code sequences in libc that were not placed there by the compiler, consider an analogy to English. English words vary in length, and there is no particular position on the page where a word must end and another start. Intel x86 code is like English written without punctuation or spaces, so that the words all run together.² The processor knows where to start reading and, continuing forward, is able to recover the individual words and make out the sentence, as it were. At the same time, one can make out more words on the page than were intentionally placed there. Some words will be suffixes of other words, as “dress” is a suffix of “address”; others will consist of the end of one word and the beginning of the next, as “head” can be found in “the address”; and so on. How frequently such things occur depends on the characteristics of the language in question, what we call its *geometry*. And the x86 ISA is extremely dense, meaning that a random byte stream can be interpreted as a series of valid instructions with high probability [3]. Thus for x86 code it is quite easy to find not just unintended words but entire unintended sequences of words. For a sequence to be potentially useful in our attacks, it must simply end in a return instruction, represented by the byte `c3`. In analyzing a large body of code such as libc we therefore expect to find many such sequences, a claim that we codify as this paper’s thesis:

Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

By contrast, on an architecture such as MIPS where all instructions are 32 bits long and 32-bit aligned there is no ambiguity about where instructions start or stop, and no unintended instructions of the sort we describe.

¹One reason is that shell parsing rules are baroque and it is consequently difficult to tell what, exactly, the shell will do with its input, making `system` difficult to use securely [32, Section 8.3]

²... if English were a prefix-free code, to be pedantic.

One way to weaken our attack is to bring the same features to the x86 architecture. McCamant and Morrisett, as part of their x86 Software Fault Isolation (SFI) design [18], propose an instruction alignment scheme that does this. However, their scheme has some downsides: first, code compiled for their scheme cannot call libraries not so compiled, so the switch must be all-or-nothing; second, their “`andl $0xffffffff, (%esp); ret`” idiom imposes a data dependency that may introduce slowdowns that are unacceptable in general-purpose software as opposed to the traditional, more limited SFI usage scenarios.³ We stress, however, that while countermeasures of this sort would impede our attack, they would not necessarily prevent it. We have taken some measures, described in Section 2, to avoid including sequences in our trie that were intentionally placed there by the compiler, but an attacker is under no such obligation, and there may well be enough sequences that are suffixes of functions in `libc` to mount our attack.

In relying intimately on the details of the x86 instruction set, our paper is inspired by two others: rix’s *Phrack* article showing how to construct alphanumeric x86 shellcode [23] and Sovarel, Evans, and Paul’s “Where’s the FEEB?,” which showed how to defeat certain kinds of instruction set randomization on the x86 [29].

How We Find Sequences. In Section 2, we describe an efficient algorithm for static analysis of x86 executables and libraries. In the version of `libc` we examined, our tool found thousands of sequences, from which we chose a small subset by means of which to mount our attack. Static analysis has recently found much use as an attack tool. For example, Kruegel et al. [15] use sophisticated symbolic execution to find ways by which an attacker can regain control after supposedly restoring a program to its pristine state, with the goal of defeating host-based intrusion detection system. In their setting, unlike ours, the attacker can execute arbitrary injected code. Their static analysis techniques, however, might be applicable to our case as well.

How We Use Sequences in Crafting an Attack. The way we interact with `libc` in return-oriented programming differs from the way we interact with `libc` in traditional return-into-`libc` attacks in three ways that make crafting gadgets a delicate, difficult task.

1. The code sequences we call are very short—often two or three instructions—and, when executed by the processor, perform only a small amount of work. In traditional return-into-`libc` attacks, the building blocks are entire functions, which each perform substantial tasks. Accordingly, our attacks are crafted at a lower level of abstraction, like assembler instead of a high-level language.
2. The code sequences we call generally have neither function prologue nor function epilogue, and are not chained together during the attack in the standard ways described in the literature, e.g., by Nergal [19].
3. Moreover, the code sequences we call, considered as building blocks, have haphazard interfaces; by contrast, the function-call interface is standardized as part of the ABI.

(Recall that there is, of course, a fourth difference between our code sequences and `libc` functions that is what makes our attack attractive: the code sequences we call weren’t intentionally placed in `libc` by the authors, and are not easily removed.) In Section 3, we show, despite the difficulties, how to construct *gadgets*—short blocks placed on the stack that chain several of instruction sequences together—that perform all the tasks

³Things would be better if Intel added 16-byte-aligned versions of `ret`, `call`, `jmp`, and `jcc` to the x86 instruction set.

one needs to perform. We describe gadgets that perform load/store, arithmetic and logic, control flow, and system call operations.

We stress that while we choose to use certain code sequences in the gadgets in Section 3, we could have used other sequences, perhaps less conveniently; and while our specific code sequences might not be found in a libc on another platform, other code sequences will be, and gadgets similar to ours could be constructed with those — at least if our thesis holds.

Previous Uses of Short Sequences in Attacks. Previous return-into-libc attacks have also used short code snippets in libc, notably Nergal’s “pop-ret” sequences [19] and the “register spring” technique introduced by dark spyrit [6] and discussed by Crandall, Wu, and Chong [5]. Our attack differs in two major ways from such previous uses:

1. Whereas previous attacks used a handful of such snippets as glue in combining the invocations of functions in libc or in jump-starting the execution of attacker-injected code, our attack uses only short sequences, and many of them, in accomplishing its goals.
2. Whereas previous attacks looked for certain prespecified snippets (e.g., call %ebx), our attack begins by enumerating the available code sequences and building an exploit from those; it would therefore be more difficult to disable our attack by eliminating particular sequences from libc.

Wait, What about Zero Bytes? The careful reader will observe that some of the gadgets we describe in Section 3 require that a NUL byte be placed on the stack. This means that they cannot be used in the payload of a simple stack-smash buffer overflow. This is not a problem, however, for the following reasons:

1. We have not optimized our gadgets to avoid NUL bytes. If they are a concern, it should be possible to eliminate the use of many of them, using the same techniques used in standard shellcode construction. For example, loading an immediate 0 into %eax could be replaced by a code sequence of the form xor %eax, %eax; ret, or by a load of 0xffffffff followed by an increment. If the address of a code sequence includes a NUL byte, we could have GALILEO choose another instance of that sequence whose address does not include a NUL byte, or we can substitute a different sequence.
2. There are other ways by which an attacker can overwrite the stack than standard buffer overflows, and not all suffer from the same constraints. For example, there is no problem writing NUL bytes onto the stack in a format-string exploit.
3. We view our techniques not in isolation but as adding to the toolbox available for return-into-libc attacks. This toolbox already contains techniques for patching up NUL bytes — as described, for example, by Nergal [19, Section 3.4] — that are just as applicable to exploits structured in the ways we describe.

A similar argument applies to the interaction of our gadgets with ASLR. Those gadgets that do not require knowledge of addresses on the stack can be used directly in the Shacham et al. [25] derandomization framework. Some of those gadgets that do require knowledge of addresses on the stack could likely be rewritten not to require it.

Our Libc Testbed. We carry out our experiments on the GNU C Library distributed with Fedora Core Release 4: libc-2.3.5.so. Our testing environment was a 2.4GHz Pentium 4 running Fedora Core Release 4, with Linux kernel version 2.6.14 and GNU libc 2.3.5, as noted.

2 Discovering Instruction Sequences in Libc

In this section, we describe our algorithm for discovering useful code sequences in libc. We sifted through the sequences output by this algorithm when run on our testbed libc to select those sequences employed in the gadgets described in Section 3.

Before we describe the algorithm, we must first make more precise our definition of “useful code sequence.” We say that a sequence of instructions is *useful* if it could be used in one of our gadgets, that is, if it is a sequence of valid instructions ending in a `ret` instruction and such that that none of the instructions causes the processor to transfer execution away, not reaching the `ret`. (It is the `ret` that causes the processor to continue to the next step in our attack.) We say that a useful sequence is *intended* if the instructions were actually inserted by the compiler in giving the machine-code compiled equivalent for some function in libc. In accordance with our thesis, the algorithm we describe attempts to avoid intended code sequences, though it does not shy away from using intended `rets` at the end of sequences.

Two observations guide us in the choice of a data structure in which to record our findings. First, any suffix of an instruction sequence is also a useful instruction sequence. If, for example, we discover the sequence “`a; b; c; ret`” in libc, then the sequence “`b; c; ret`” must of course also exist. Second, it does not matter to us how often some sequence occurs, only that it does.⁴ Based on these observations, we choose to record sequences in a trie. At the root of the trie is a node representing the `ret` instruction; the “child-of” relation in the trie means that the child instruction immediately precedes the parent instruction at least once in libc. For example, if, in the trie, a node representing `pop %eax` is a child of the root node (representing `ret`) we can deduce that we have discovered, somewhere in libc, the sequence `pop %eax; ret`.

Our algorithm for populating the trie makes use of following fact: It is far simpler to scan *backwards* from an already found sequence than to disassemble forwards from every possible location in the hope of finding a sequence of instructions ending in a `ret`. When scanning backwards, the sequence-so-far forms the suffix for all the sequences we discover. The sequences will then all start at instances of the `ret` instruction, which we can scan libc sequentially to find.

In looking backwards from some location, we must ask: Does the single byte immediately preceding our sequence represent a valid one-byte instruction? Do the two bytes immediately preceding our sequence represent a valid two-byte instruction? And so on, up to the maximum length of a valid x86 instruction.⁵ Any such question answered “yes” gives a new useful sequence of which our sequence-so-far is a suffix, and which we should explore recursively by means of the same approach. Because of the density of the x86 ISA, more than one of these questions can simultaneously have a “yes” answer.⁶

We now present our algorithm — which we call GALILEO — in pseudocode.

Algorithm GALILEO:

```
create a node, root, representing the ret instruction;
place root in the trie;
for pos from 1 to textseg_len do:
    if the byte at pos is c3, i.e., a ret instruction, then:
        call BUILDFROM(pos, root).
```

⁴From all the occurrences of a sequence, we might prefer to use one whose address does not include a NUL byte over one that does.

⁵Including all instruction-modifying prefixes, 20 bytes.

⁶In fact, amongst the useful sequences we discover in libc there is a point where four valid instructions all end at the same point; and, examining libc as a whole, there is a point where seven valid instructions do.

```

Procedure BUILDFROM(index pos, instruction parent_insn):
  for step from 1 to max_insn_len do:
    if bytes [(pos - step) ... (pos - 1)] decode as a valid instruction insn then:
      ensure insn is in the trie as a child of parent_insn;
      if insn isn't boring then:
        call BUILDFROM(pos - step, insn).

```

“Boring” Instructions. The definition of “boring” we use is the following:

1. the instruction is a leave instruction and is followed by a ret instruction; or
2. the instruction is a pop %ebp instruction and is immediately followed by a ret instruction; or
3. the instruction is a return or an unconditional jump.

The last of these criteria eliminates instruction streams in which control transfers elsewhere before the ret is reached, as these are useless for our purposes. The other two are intended to capture, and allow us to ignore, instruction streams that are actually generated by the compiler. Because the libc we examined was compiled with frame pointer enabled, functions in libc will, by and large, end either with a “leave; ret” sequence or an equivalent where the leave instruction is replaced by mov and pop instructions.

It is important to observe that the conditions given here eliminate instruction sequences that would be useful in crafting exploits. There are three ways in which they do so. First, even if we wish to avoid calling actual functions in libc, suffixes of those functions might prove useful and, if short, difficult for the compiler-writer to eliminate. Second, the same characteristics that allow us to discover unintended instruction sequences elsewhere will also allow us to discover, within the body of libc functions, unintended sequences that end in intended “leave; ret” sequences. Third, both leave and pop %ebp are one-byte instructions, and it is possible that a “leave; ret” sequence we come upon wasn’t intended at all, but is found in the libc byte stream in the same way that unintended rets are, explained in Section 4. Note that while the techniques we develop for generating programs from chains of instruction sequences do not usually interact with leaves, it is possible to modify our techniques to work in this setting using the frame-chaining methods described by Nergal [19]. That we are able to mount our attacks even without using the code snippets eliminated by the conditions above gives further evidence, of course, for our thesis.

Implementation and Timing. Our implementation of GALILEO follows the pseudocode given above quite closely. To discover what portion of libc is mapped as an executable segment, our code parses libc’s ELF headers. We make use of two helper libraries. To parse the ELF headers, we use GNU libelf, version 0.8.9 [22]; to decode x86 instructions, we use the Bastard project’s libdisasm [17], version 0.21-pre from CVS, with some local modifications. Analyzing the 1,189,501 bytes of libc’s executable segment yields a trie with 15,121 nodes, and takes 1.6sec on a 1.33GHz PowerPC G4 with 1GB RAM. While it should be possible to improve the running time of the algorithm—for example, by using memoization to avoid decoding a particular byte sequence in libc several times—we judged our implementation’s performance to be already quite adequate.

3 Principles of Return-Oriented Programming

This section is intended to serve as a catalogue for the actions that we can perform using the sequences we find in libc, and as a tutorial to return-oriented programming generally. Accordingly, we provide more explanatory detail for the earlier gadgets than the later.

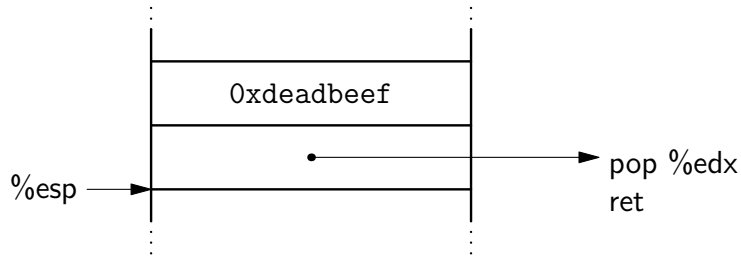


Figure 1: Load the constant 0xdeadbeef into %edx.

Each of our gadgets expects to be entered in the same way: the processor executes a `ret` with the stack pointer, `%esp`, pointing to the bottom word of the gadget. This means that, in an exploit, the first gadget should be placed so that its bottom word overwrites some function’s saved return address on the stack. Further gadgets can be placed immediately after the first or, by means of the control flow gadgets given in Section 3.3, in arbitrary locations. (It is helpful for visualizing gadget placement to think of the gadgets as being instructions in a rather strange computer.)

3.1 Load/Store

We consider three cases: loading a constant into a register; loading the contents of a memory location into a register; and writing the contents of a register into a memory location.

Loading a Constant. The first of these can trivially be accomplished using a sequence of the form `pop %reg; ret`. One such example is illustrated in Figure 1. In this figure as in all the following, the entries in the ladder represent words on the stack; those with larger addresses are placed higher on the page. Some words on the stack will contain the address of a sequence in `libc`. Our notation for this shows a pointer from the word to the sequence. Other words will contain pointers to other words, or immediate values. In the example here, once the processor is executing the sequence `pop %edx; ret`, the `ret` that caused it to enter the gadget will also have caused `%esp` to be incremented by a word; the `pop %edx` instruction, therefore, will pop the next word on the stack — `0xdeadbeef`, in this case — into `%edx`, advancing `%esp` once more, past the end of the gadget, so that the `ret` instruction causes execution to continue with the next gadget, placed above it.

Loading from Memory. We choose to load from memory into the register `%eax`, using the sequence `movl 64(%eax), %eax; ret`. We first load the address into `%eax`, using, for example, the constant-load procedure detailed above. Because of the immediate offset in the `movl` instruction we use, the address in `%eax` must actually be 64 bytes less than the address we wish to load. We then apply the `movl` sequence, after which `%eax` contains the contents of the memory location. The procedure is detailed in Figure 2. Note the notation we use to signify, “the pointer in this cell requires that 64 be added to it so that it points to some other cell.”

Storing to Memory. We use the sequence `movl %eax, 24(%edx); ret` to store the contents of `%eax` into memory. We load the address to be written into `%edx` using the constant-load procedure above. The procedure is detailed in Figure 3.

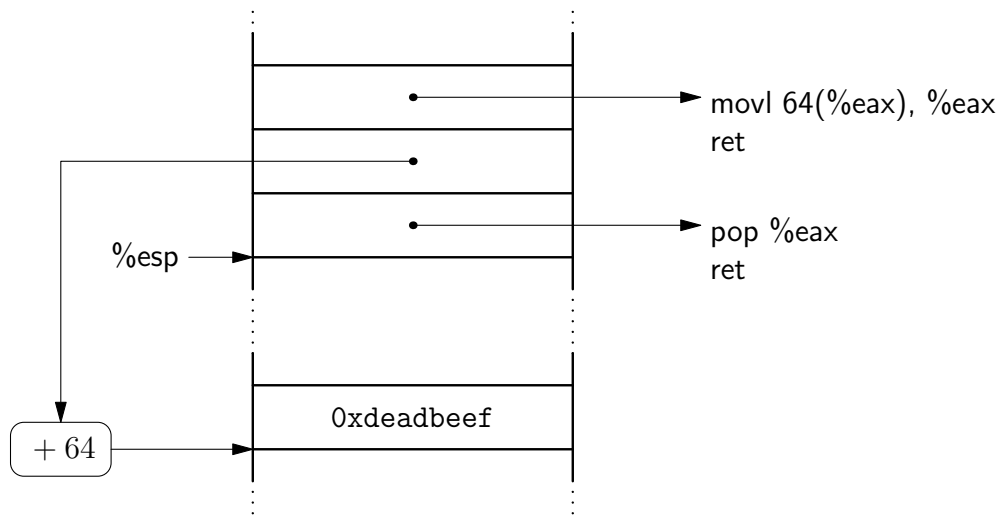


Figure 2: Load a word in memory into `%eax`.

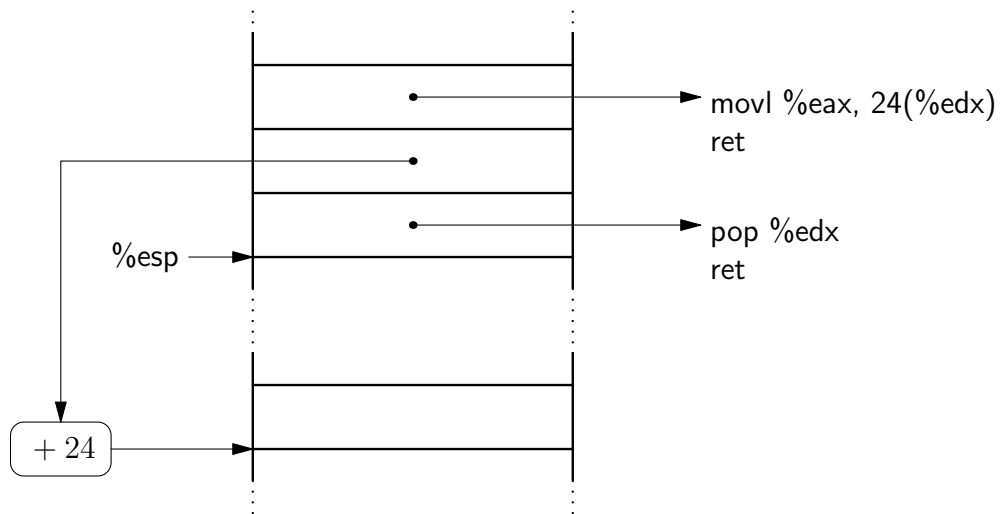


Figure 3: Store `%eax` to a word in memory.

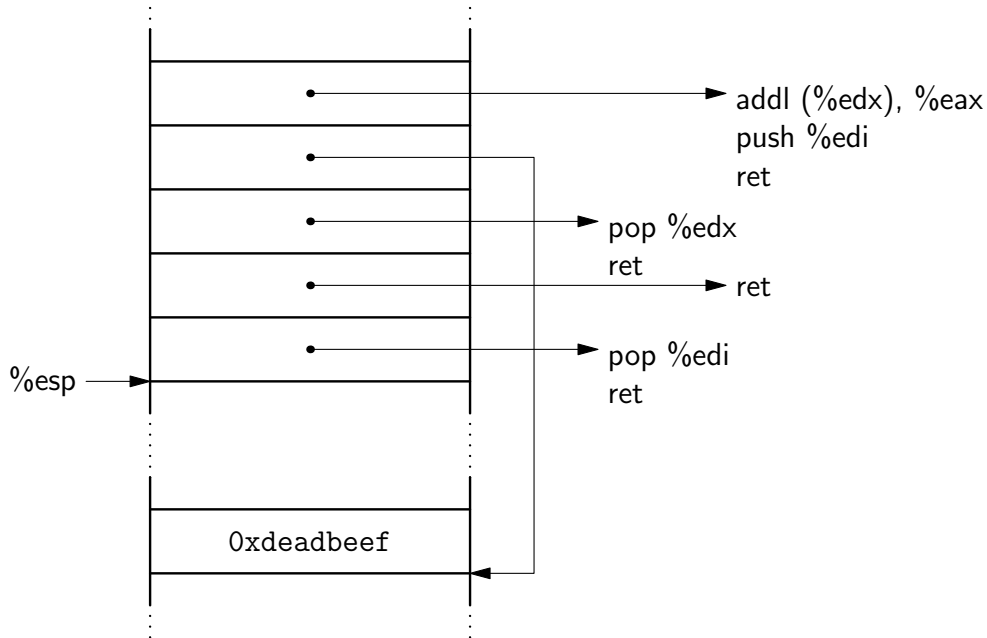


Figure 4: Simple add into `%eax`.

3.2 Arithmetic and Logic

There are many approaches by which we could implement arithmetic and logic operations. The one we choose, which we call our *ALU paradigm*, is as follows. For all operations, one operand is `%eax`; the other is a memory location. Depending on what is more convenient, either `%eax` or the memory location receives the computed value. This approach allows us to compute memory-to-memory operations in a simple way: we load one of the operands into `%eax`, using the load-from-memory methods of Section 3.1; we apply the operation; and, if the result is now held in `%eax`, we write it to memory, using the store-to-memory methods of the same section.

Below, we describe in detail some of the operations we implement — particularly those that introduce new techniques — and the remainder more briefly.

Add. The most convenient sequence for performing an add that fits into our ALU paradigm is the following:

```
addl (%edx), %eax;  push %edi;  ret.      (1)
```

The first instruction adds the word at `%edx` to `%eax`, which is exactly what we want. The next instruction, however, creates some problems. Whereas a “popret” sequence is convenient for implementing a constant-load operation, a “pushret” sequence is inconvenient for two reasons. First, the value pushed onto the stack is then immediately used by the `ret` instruction as the address for the next code sequence to execute, which means the values we can push are restricted. Second, the push overwrites a word on the stack, so that if we execute the gadget again (say, in a loop) it will not behave the same.

We first present a simple approach that does not take the second problem into account. Before undertaking the `addl` instruction sequence, we load into `%edi` the address of a `ret` instruction. In return-oriented programming, a `ret` acts like `nop`, increasing `%esp` but otherwise having no effect. We illustrate this version

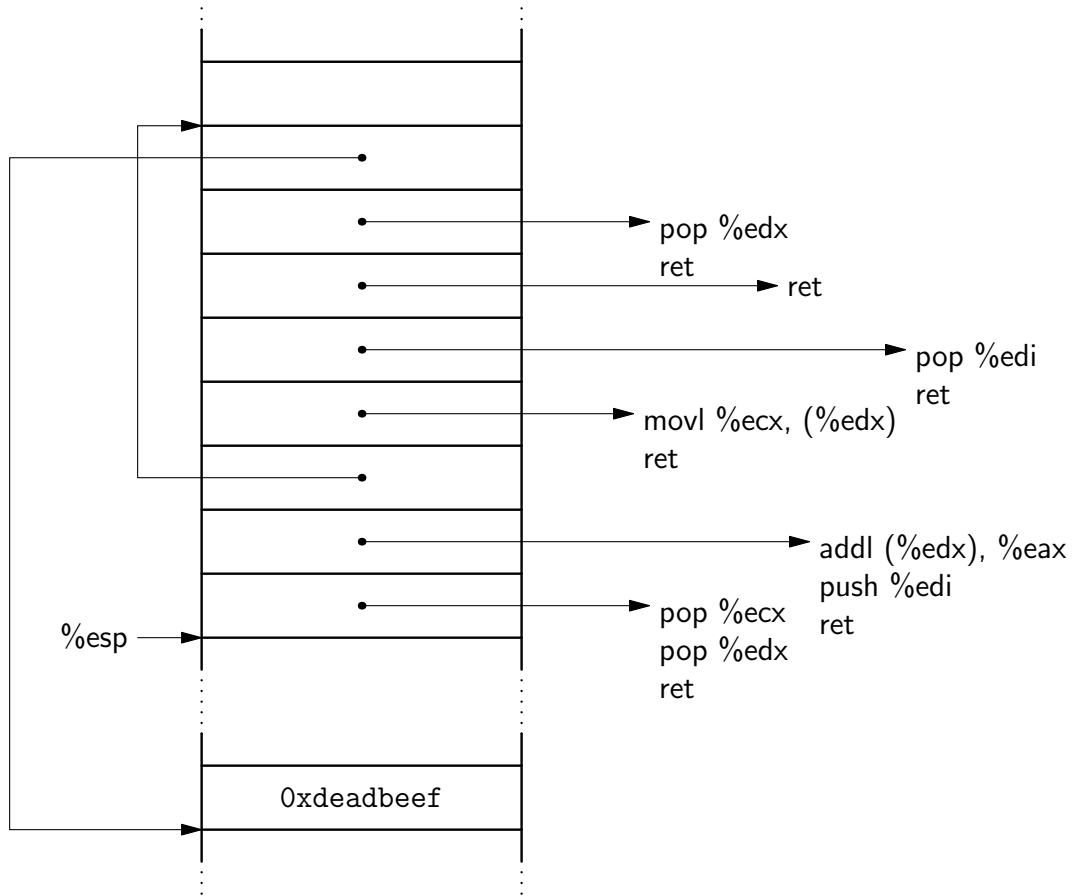


Figure 6: Repeatable add into `%eax`.

The immediate offset in the `xorb` instruction means that the values we load into `%ebx` must be adjusted appropriately. The `and` and `or` operations have the effect of destroying the value in `%al`, but by then we have already used `%al`, so this is no problem. (If we want to undertake another operation with the value in `%eax`, we must reload it from memory.) The `push` operation means that we must load into `%ebp` the address of a `ret` instruction and that, if we want the `xor` to be repeatable, we must rewrite the `xorb` instructions into the gadget each time, as described for repeatable addition above. Figure 7 gives the details for a (one-time) `xor` operation.

And, Or, Not. Bitwise-`and` and `-or` are also best implemented using bitwise operations, in a manner quite similar to the `xor` method above. The code sequences are, respectively,

```
andb %al, 0x5d5e0cc4(%ebx); ret    and    orb %al, 0x40e4602(%ebx); ret.
```

These code sequences have fewer side effects than (2) for `xor`, above, so they are simpler to employ. Bitwise-`not` can be implemented by xoring with the all-1 pattern.

Shifts and Rotates. We first consider shifts and rotates by an immediate value. In this case, instead of implementing the full collection of shifts and rotates, we implement a single operation: a left rotate, which suffices for constructing the rest: a right rotate by k bits is a left rotate by $32 - k$ bits; a shift by k bits in either direction is a rotate by k bits followed by a mask of the bits to be cleared, which can itself be computed using the bitwise-`and` method discussed above. The code sequence we use for rotation is `roll %cl, 0x17383f8(%ebx); ret`. The corresponding gadget is detailed in Figure 8.

We now consider shifts and rotates by a variable number of bits. The gadget in Figure 8 reads the value of `%ecx` from the stack. If we wish for this value to depend on some other memory location, we can simply read that memory location and write it to the word on the stack from which `%ecx` is read. Implementing variable-bit shifts is a bit more difficult, because we must now come up with the mask corresponding to the shift bits. The easiest way to achieve this is to store a 32-word lookup table of masks in the program.

3.3 Control Flow

Unconditional Jump. Since in return-oriented programming the stack pointer `%esp` takes the place of the instruction pointer in controlling the flow of execution, an unconditional jump requires simply changing the value of `%esp` to point to a new gadget. This is quite easy to do using the instruction sequence `pop %esp; ret`. Figure 9 shows a gadget that causes an infinite loop by jumping back on itself.

Loops in return-into-libc exploits have been considered before: see Gera’s “esoteric #2” challenge [9].

Conditional Jumps. These are rather more tricky. Below we develop a method for obtaining conditional jumps.

To begin, some review. The `cmp` instruction compares its operands and, based on their relationship, sets a number of flags in a register called `%eflags`. In x86 programming, it is often unnecessary to use `cmp` directly, because many operations set flags as a side effect. The conditional jump instructions, `jcc`, cause a jump when the flags satisfy certain conditions. Because this jump is expressed as a change in the instruction pointer, the conditional jump instructions are not useful for return-oriented programming: What we need is a conditional change in the *stack* pointer.

The strategy we develop is in three parts, which we tackle in turn:

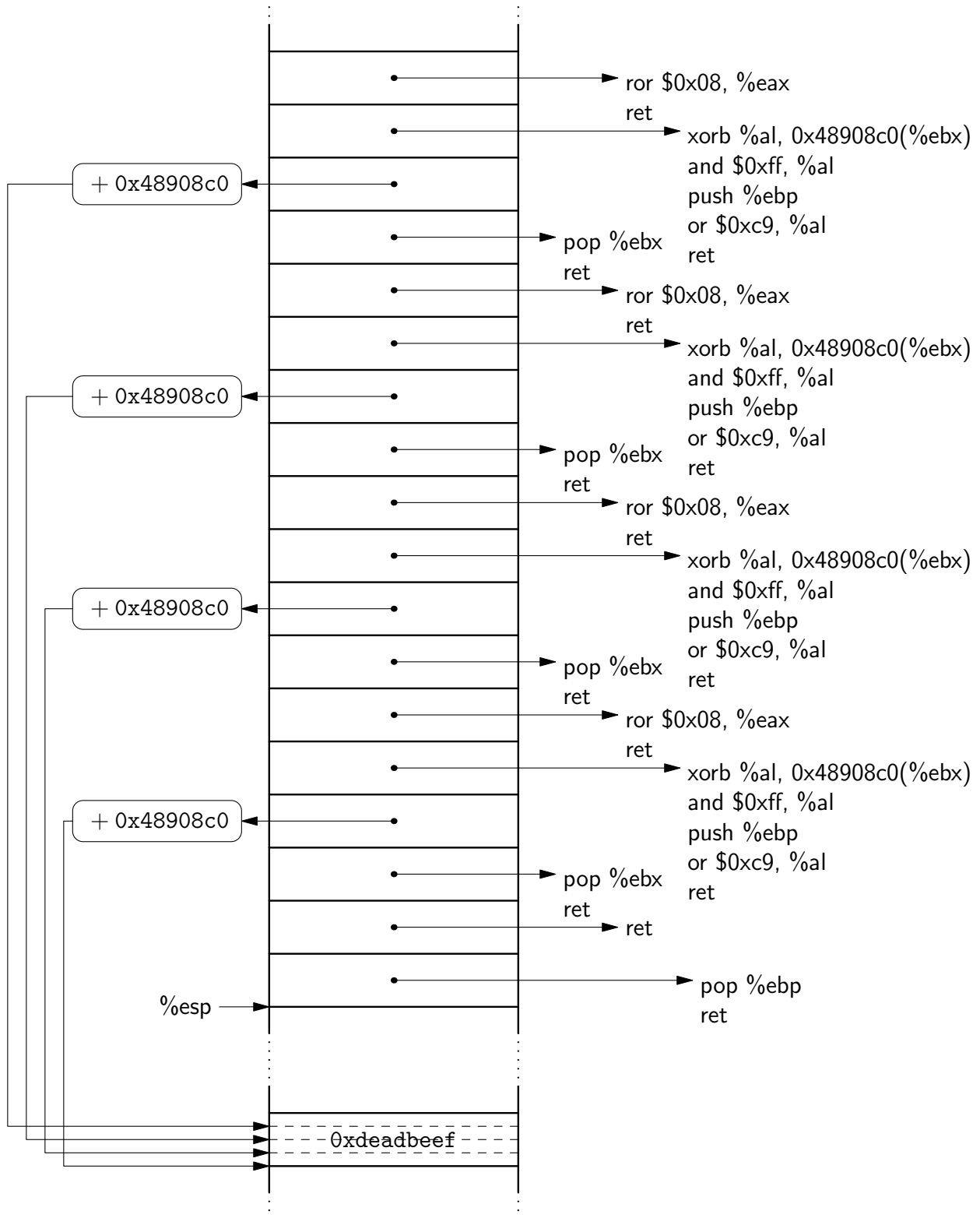


Figure 7: Exclusive or from %eax.

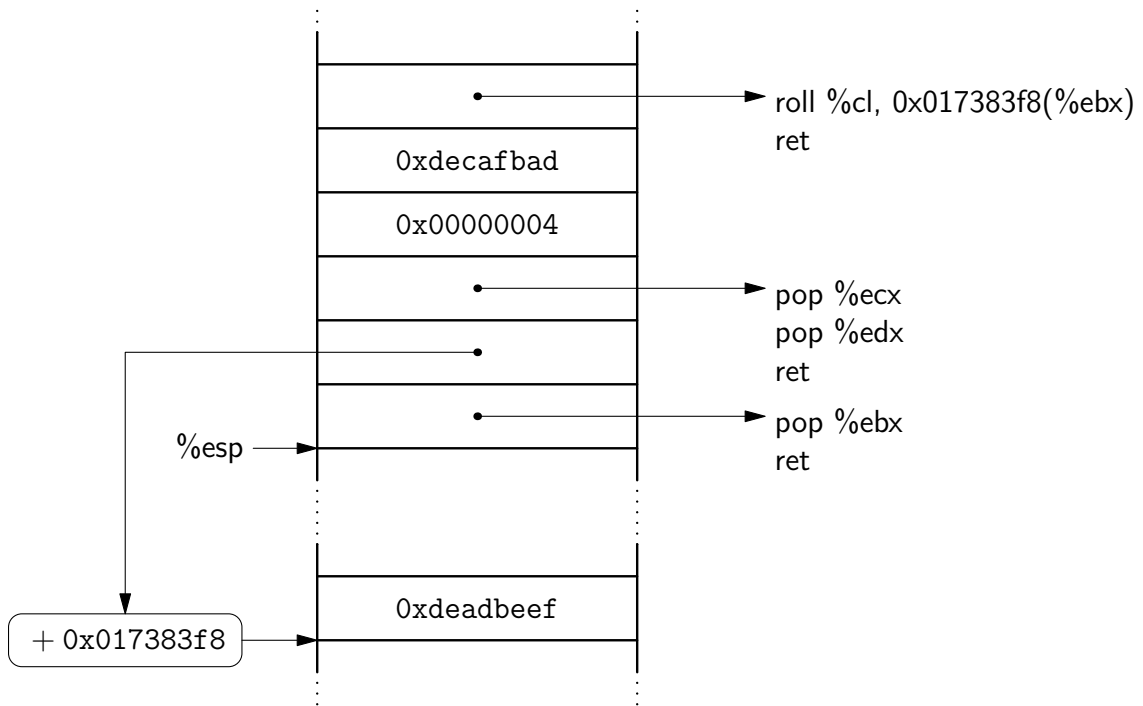


Figure 8: Immediate rotate, 4 bits leftward, of memory word.

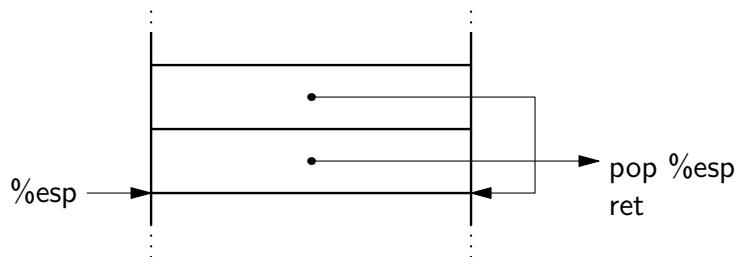


Figure 9: An infinite loop by means of an unconditional jump.

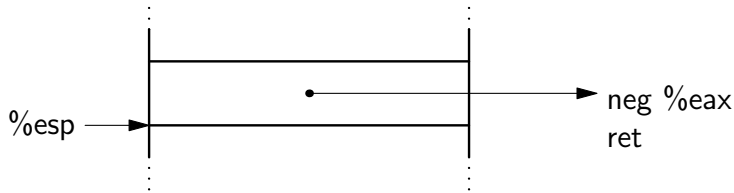


Figure 10: Conditional jumps, phase one: Clear CF if %eax is zero, set CF if %eax is zero.

1. Undertake some operation that sets (or clears) flags of interest.
2. Transfer the flags from %eflags to a general-purpose register, and isolate the flag of interest.
3. Use the flag of interest to perturb %esp conditionally by the desired jump amount.

For the first task, we choose to use the carry flag, CF, for reasons that will become clear below. Employing just this flag, we obtain the full complement of standard comparisons. Most easily, we can test whether a value is zero by applying `neg` to it. The `neg` instruction (and its variants) calculate two's-complement and, as a side effect, clears CF if its operand is zero and sets CF otherwise. Figure 10 shows the simplest case, in which the value to test is held in %eax. (Note that this is in keeping with our ALU paradigm of Section 3.2.)

If we wish to test whether two values are equal, we can subtract one from the other and test (using `neg`, as above) whether the result is zero. If we wish to test whether one value is larger than another, we can, again, subtract the first from the second; the `sub` instruction (and its variants) set CF when the subtrahend is larger than the minuend.

For the second task, the natural way to proceed is the `lahf` instruction, which stores the five arithmetic flags (SF, ZF, AF, PF, and CF) in %ah. Unfortunately, this instruction is not available to us in the `libc` sequences we found. Another way is the `pushf` instruction, which pushes a word containing all of %eflags onto the stack. This instruction is available to us, but like all `pushret` is tricky to use in a return-oriented setting.

Instead, we choose a third way. Several instructions use the carry flag, CF, as an input: in particular, left and right rotates with carry, `rcl` and `rcr`, and add with carry, `adc`. Add with carry computes the sum of its two operands and the carry flag, which is useful in multiword addition algorithms. If we take the two operands to be zero, the result is 1 or 0 depending on whether the carry flag is set — exactly what we need. This we can do quite easily by clearing %ecx and using the instruction sequence `adc %cl, %cl; ret`. The process is detailed in Figure 11. We note, finally, that we can evaluate complicated boolean expressions by collecting CF values for multiple tests and combining them with the logical operations described in Section 3.2.

For the third phase, we proceed as follows. We have a word in memory that contains 1 or 0. We transform it to contain either `esp_delta` or 0, where `esp_delta` is the amount we'd like to perturb %esp by if the condition evaluates as true. One way to do this is as follows. The two's complement of 1 is the all-1 pattern and the two's complement of 0 is the all-0 pattern, so applying `negl` to the word containing CF we have all-1s or all-0s. Then taking bitwise and of the result and `esp_delta` gives a word containing `esp_delta` or 0. This process is detailed in Figure 12. (The instruction sequences we use have some side effects that must be worked around, but the process itself is straightforward.)

Now, we have the desired perturbation, and it is simple to apply it to the stack pointer by means of the sequence

```
addl (%eax), %esp;  addb %al, (%eax);  addb %cl, 0(%eax);  addb %al, (%eax);  ret
```

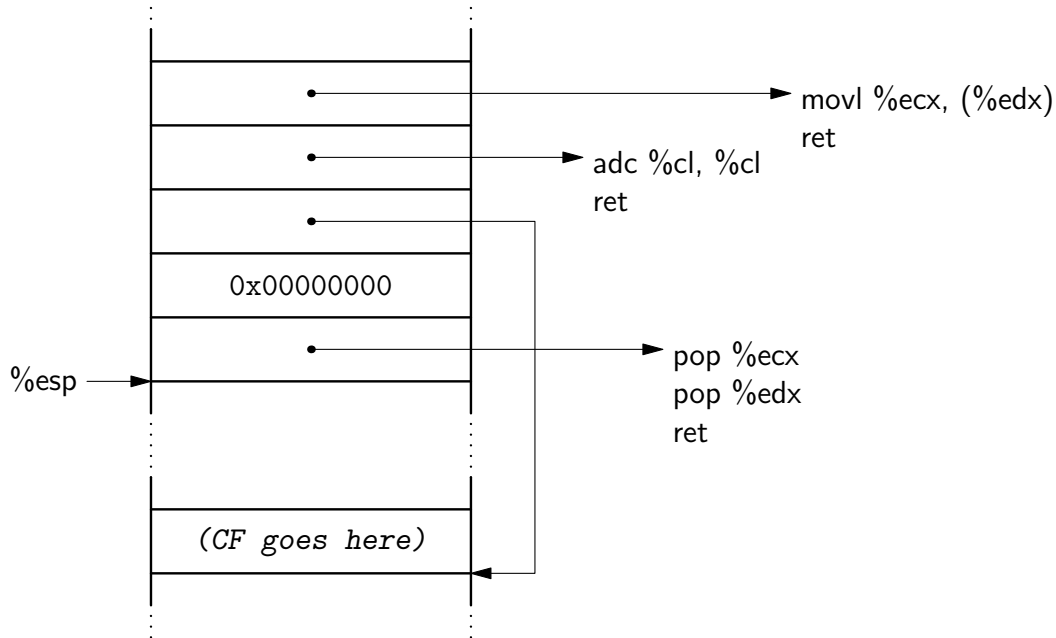


Figure 11: Conditional jumps, phase two: Store either 1 or 0 in the data word labeled “CF goes here,” depending on whether CF is set or not.

with `%eax` pointing to the displacement. The extra operations have the effect of destroying the displacement, but as it has already been used this is no problem. The procedure is detailed in Figure 13.

3.4 System Calls

Our key observation is that many system calls have simple wrappers in `libc` that behave, broadly, as follows:

1. move the arguments from the stack to registers, and set the syscall number in `%eax`;
2. trap into the kernel (indirectly through `linux-gate.so.1`); and
3. check for error and translate the return value appropriately, leaving it in `%eax`.

A typical example is the code for `umask`:

```

89 da          mov %ebx, %edx
8b 5c 24 04    movl 4(%esp), %ebx
b8 3c 00 00 00 mov $0x0000003C, %eax
65 ff 15 10 00 00 00 lcall %gs:0x10(,0)
89 d3          mov %edx, %ebx
c3            ret

```

Here the `lcall` through the GS segment invokes `__kernel_vsycall`, which itself actually issues the `sysenter` or `int 0x80` instruction (cf. [8]). If we set up the system call parameters ourselves and jump into a wrapper at step 2—that is, immediately before the `lcall` in the fourth line, for `umask`—we can invoke any system call we choose, with any arguments.

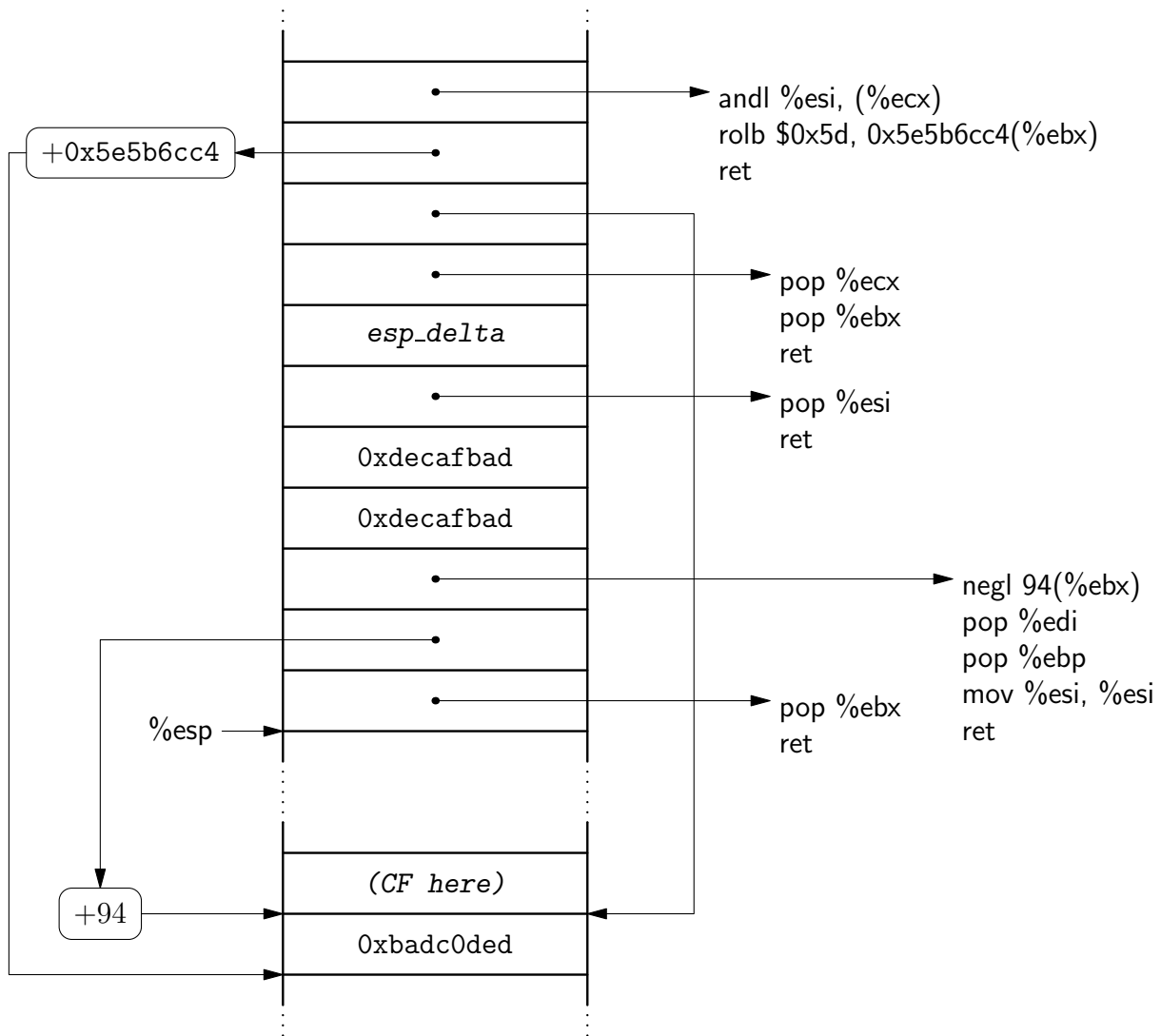


Figure 12: Conditional jumps, phase three, part one: Convert the word (labeled “CF here”) containing either 1 or 0 to contain either `esp_delta` or 0. The data word labeled `0xbadc0ded` is used for scratch.

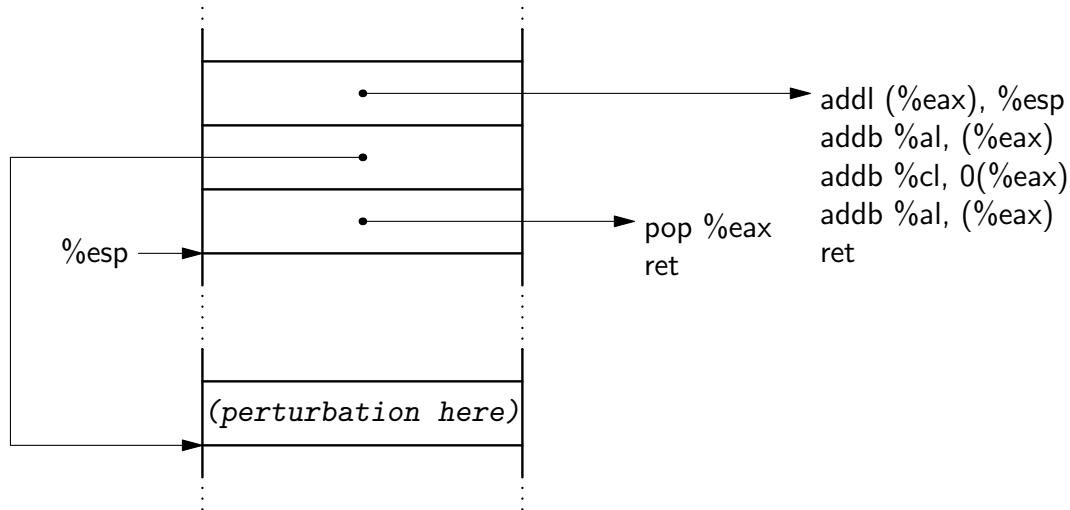


Figure 13: Conditional jumps, phase three, part two: Apply the perturbation in the word labeled “perturbation here” to the stack pointer. The perturbation is relative to the end of the gadget.

For system calls, then, we are using intended sequences in `libc`, rather than the (largely) unintended sequences elsewhere. Since nearly all useful programs make system calls, the requirement that a system call wrapper function be available is milder than the requirement that specific `libc` routines, such as `system`, be available. On Linux, we can also do away with this assumption entirely by calling `__kernel_vsyscall` directly, after finding it by parsing the ELF auxiliary vectors (cf. [7]).

We detail, in Figure 14, a gadget that invokes a system call. Arguments could be loaded ahead of time into the appropriate registers: in order, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp`. One way to load these registers is as follows. Using the techniques of Section 3.1, load the desired value into `%eax`, then write it into the second word of a gadget whose first word points to an instruction sequence of the form `pop %r32; ret`, where `r32` is the register to set.

3.5 Function Calls

Finally, we note that nothing prevents us from making calls to arbitrary functions in `libc`. This is, in fact, the basis for previous return-into-`libc` exploits, and the required techniques are described in by Nergal [19]; the discussion of “frame faking” is of particular interest. It suffices to add that it is best to invoke functions with the stack pointer set to a part of the stack not used by other return-oriented code, because otherwise those functions might, in using the stack, trash a gadget that we intend to reinvoke.

4 A Catalog of rets

In this section, we give some statistics about the origin of `c3` bytes in the `libc` executable segment. Our methodology is as follows. For each `c3` byte that we find, we check whether it is within a span of bytes belonging to a function that is exported in `libc`’s `SYMTAB` section.⁷ We then disassemble the function until we

⁷There are substantially more functions listed in the `SYMTAB` section than in the `DYNSYM` section, which lists only the functions that are actually made available for dynamic linking.

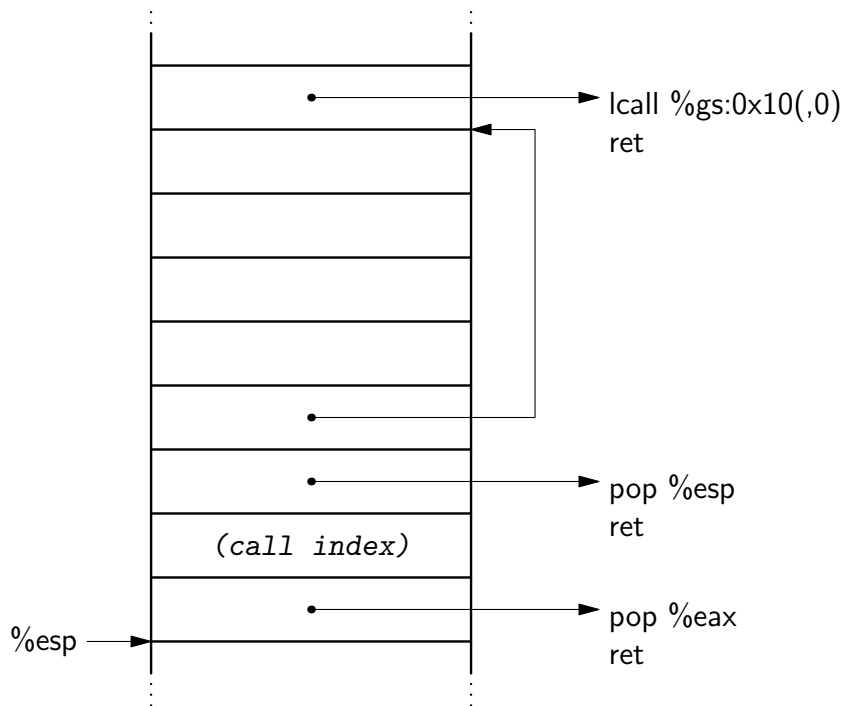


Figure 14: A system call with no arguments; the system call index is stored in the second word from bottom. Arguments could be loaded beforehand into `%ebx`, `%ecx`, etc. We leave space in case the `vsyscall` function spills values onto the stack, as the `sysenter`-based version does. Note that the word pointing to `lcall` would be overwritten also; a repeatable version of this gadget would need to restore it each time.

discover which instruction includes the `c3` byte. Not all of `libc`'s executable segment is covered by exported functions. Some of the segment is taken up by ELF headers, and some by static functions that are not named in the SYMTAB section. Nevertheless, this methodology is sufficient to allow us to draw meaningful conclusions.

Out of 975,626 covered bytes, 5,483 are `c3` bytes, or one in every 178. (This is more than the expected one-in-256 because `ret` instructions occur often.)

- 3,429 are actually `ret` instructions. Since there are only 3,157 entrypoints listed in the SYMTAB section, this means that some functions have more than one return instruction.
- 1,692 occur in the ModR/M byte for an `add imm32, %ebx` instruction, opcode `81 c3 imm32`. Immediate-`add` is part of “Immediate Grp 1” (opcodes 80–83), which use bits 3–5 of the ModR/M byte to encode an opcode extension. In this case bits 6 and 7 (11) specify that the target is a register; bits 0–2 (011) name `%ebx`; and bits 3–5 (000) specify an `add` operation. For comparison, `81 c2` would encode `add imm32, %edx`, and `81 cb` would encode `or imm32, %ebx`. See Tables 2-1 and A-4 of [12].
- 290 occur in immediate displacements. Of these, 273 specify offsets to the instruction point — 109 relative calls, 100 relative conditional jumps, and 64 relative unconditional jumps — and the other 17 specify data offsets, as in `movb %al, -61(%ebp)`, opcode `88 45 c3`.
- 35 occur in a proper ModR/M byte, indicating `%eax` and `%ebx` as source and target, respectively. Of these, 33 are in the instruction `add %eax, %ebx`, opcode `89 c3`, and the other two are `shrd %cl, %eax, %ebx` and `shld %cl, %eax, %ebx`, opcodes `0f ad c3` and `0f a5 c3`.
- 28 occur in immediate constants, in `add`, `mov`, and `movw` instructions.
- 8 occur in the SIB byte, indicating addressing of the form `(%ebx,%eax,8)`. These all happen to be in instances of the instruction `movl r/m32, r32` in which the ModR/M byte specifies `SIB+disp32` addressing. (Opcodes: `8b modr/m c3 imm32`, with `modr/m` being of the form `10bbb100`.)
- 1 occurs in the floating point operation `fld %st(3)`, opcode `d9 c3`. (More generally, `d9 c0+i` encodes `fld %st(i)`.)

Can we avoid spurious rets? Some modest changes to GCC might yield a `libc` without unintended `c3` bytes. For example, each procedure could have only a single exit point (with the standard `leave`; `ret` sequence), to which early exits could jump. The `%ebx` register could be avoided as an accumulator for adds. Moves from `%eax` to `%ebx` could be avoided or written using instructions other than `mov`. Instruction placement could be jiggered — in most cases, at least — to avoid offsets with `c3` bytes in them.

Such a strategy might indeed succeed in ridding generated executables of unintended `c3` bytes. The cost would be a compiler that is less transparent and more complicated, and a certain loss of efficiency in the use of registers on an already register-starved architecture: `%ebx` is handy as an accumulator because, unlike `%eax`, `%ecx`, and `%edx`, it is callee-saved in the Intel calling convention [31].

We must be clear, however, that while this would eliminate unintended `rets`, it would not eliminate unintended sequences of instructions that end in a `ret`. This is because whereas the attacker is now constrained to choosing attack strings that are suffixes of valid `libc` functions, he still need not begin his strings on an intended instruction boundary. Thus, for example, the `libc` entrypoint `svc_getreq` ends with the following cleanup code:

```

81 c4 88 00 00 00      add $0x00000088, %esp
5f                      pop %edi
5d                      pop %ebp
c3                      ret

```

Taking the last four bytes of this, the adversary instead obtains

```

00 5f 5d      addb %bl, 93(%edi)
c3           ret

```

There is a more fundamental problem, however. The gadgets we described in Section 3 made use only of instruction sequences ending in `c3` bytes because these were sufficient. However, the x86 ISA actually includes *four* opcodes that perform a return instruction: `c3` (near return), `c2` *imm16* (near return with stack unwind), `cb` (far return), and `ca` *imm16* (far return with stack unwind). The variants with stack unwind, having popped the return address off the stack, increment the stack pointer by *imm16* bytes; this is useful in calling conventions where arguments are callee-cleaned. The far variants pop `%cs` off the stack as well as `%eip`. These three variants are more difficult to use in exploits of the sort we describe. For the far variants, the correct code segment must be placed on the stack; for the stack-unwind variants, a stack underflow must be avoided. Nevertheless, it should be possible to use them. And eliminating instances of all four would be difficult, as it would require avoiding four byte values out of 256.

Moreover, if we have the ability to load immediate values into registers, for example using the techniques of Section 3.1, then we can use some sequences that do not end in a `ret`. For example, if `%ebx` points to a `ret` instruction in `libc`, then any sequence ending in `jmp %ebx` can be used. This is simply register springs (cf. [6, 5]) in a return-into-`libc` context. With a bit more setup, we can also use sequences ending in `jmp imm(%esp)`, if the word at `imm(%esp)` contains the address of a `ret`, and again with other registers replacing `%esp`. This translates to the return-into-`libc` context a technique due to Litchfield [16].

Finally, we note that the `libc` executable image includes areas that are not intended as executable code: notably, the ELF headers. These might contain return instructions as well, which modifying the compiler will not address.

5 Conclusion and Future Work

We presented a new way of organizing return-into-`libc` exploits on the x86 that is fundamentally different from previous techniques. By means of static analysis we discovered short instruction sequences; we then showed how to combine such sequences into gadgets that allow an attacker to perform arbitrary computation. There are several directions for future work.

We developed a C-language tool, called *Delilah*, that allows us to generate an exploit string by combining a number of gadgets. We intend to turn *Delilah* into an assembler whose input is a simple assembly for some virtual computer and whose output is a return-oriented exploit. A more ambitious project would add a return-oriented backend to `GCC` or `LLVM`.

To build our gadgets we combed over the output of `GALILEO` manually. It should be possible, however, to analyze the available code sequences automatically to discover how to combine them into gadgets.

A second research direction would attempt to validate (or invalidate) our thesis by examining C libraries in other platforms. While the gadgets we describe all derive from a particular distribution of `GNU libc`, the techniques we present for discovering sequences and combining them into gadgets should be universally applicable. A preliminary analysis we conducted of `msvcrt.dll`, the Microsoft C runtime, seemed promising.

Acknowledgments

The authors thank Dan Boneh, Eu-Jin Goh, Nagendra Modadugu, Eric Rescorla, Mike Sawka, Nick Vossbrink, and Robert Zimmerman for helpful discussions regarding this work; Avram Shacham for his detailed comments on the manuscript; and members of the MIT Cryptography and Information Security Seminar, Berkeley Systems Lunch, and Stanford Security Lunch for their comments on early presentations of this work.

References

- [1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), Nov. 1996. <http://www.phrack.org/archives/49/P49-14>.
- [2] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), Aug. 2001. <http://www.phrack.org/archives/57/p57-0x09>.
- [3] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Info. & System Security*, 8(1):3-40, Feb. 2005.
- [4] blexim. Basic integer overflows. *Phrack Magazine*, 60(10), Dec. 2002. <http://www.phrack.org/archives/60/p60-0x0a>.
- [5] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In K. Julisch and C. 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), Sept. 1999. <http://www.phrack.org/archives/55/P55-15>.
- [7] M. Garg. About ELF auxiliary vectors, Aug. 2006. Online: <http://manugarg.googlepages.com/aboutelfauxiliaryvectors>.
- [8] M. Garg. Sysenter based system call mechanism in Linux 2.6, July 2006. Online: http://manugarg.googlepages.com/systemcallinlinux2_6.html.
- [9] Gera. Insecure programming by example. Online: <http://community.corest.com/~gera/InsecureProgramming/>.
- [10] gera and riq. Advances in format string exploiting. *Phrack Magazine*, 59(7), July 2001. <http://www.phrack.org/archives/59/p59-0x07>.
- [11] O. Horovitz. Big loop integer protection. *Phrack Magazine*, 60(9), Dec. 2002. <http://www.phrack.org/archives/60/p60-0x09>.
- [12] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 2001.
- [13] M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), Aug. 2001. <http://www.phrack.org/archives/57/p57-0x08>.

- [14] klog. The frame pointer overwrite. *Phrack Magazine*, 55(8), Sept. 1999. <http://www.phrack.org/archives/55/P55-08>.
- [15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In P. McDaniel, editor, *Proc. 14th USENIX Sec. Symp.*, pages 161–76. USENIX, Aug. 2005.
- [16] D. Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, Sept. 2003. Online: <http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf>.
- [17] mammon.. The Bastard project: libdisasm. <http://bastard.sourceforge.net/libdisasm.html>.
- [18] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In A. Keromytis, editor, *Proc. 15th USENIX Sec. Symp.*, pages 209–24. USENIX, July 2006.
- [19] Nergal. The advanced return-into-lib(c) exploits (PaX case study). *Phrack Magazine*, 58(4), Dec. 2001. <http://www.phrack.org/archives/58/p58-0x04>.
- [20] PaX Team. PaX address space layout randomization (ASLR). <http://pax.grsecurity.net/docs/aslr.txt>.
- [21] PaX Team. PaX non-executable pages design & implementation. <http://pax.grsecurity.net/docs/noexec.txt>.
- [22] M. Riepe. GNU Libelf. <http://www.mr511.de/software/>.
- [23] rix. Writing ia32 alphanumeric shellcodes). *Phrack Magazine*, 57(15), Dec. 2001. <http://www.phrack.org/archives/58/p57-0x0f>.
- [24] Scut/team teso. Exploiting format string vulnerabilities. <http://www.team-teso.net>, 2001.
- [25] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In B. Pfitzmann and P. Liu, editors, *Proc. 11th ACM Conf. Comp. and Comm. Sec. — CCS 2004*, pages 298–307. ACM Press, Oct. 2004.
- [26] Solar Designer. StackPatch. <http://www.openwall.com/linux>.
- [27] Solar Designer. “return-to-libc” attack. Bugtraq, Aug. 1997.
- [28] Solar Designer. JPEG COM marker processing vulnerability in Netscape browsers, July 2000. Online: <http://www.openwall.com/advisories/OW-002-netscape-jpeg/>.
- [29] N. Sovarel, D. Evans, and N. Paul. Where’s the FEEB? the effectiveness of instruction set randomization. In P. McDaniel, editor, *Proc. 14th USENIX Sec. Symp.*, pages 145–60. USENIX, Aug. 2005.
- [30] The Metasploit Project. Shellcode archive. Online: <http://www.metasploit.com/shellcode.html>.
- [31] The Santa Cruz Operation. *System V Application Binary Interface: Intel386 Architecture Processor Supplement*, fourth edition, 1996.

- [32] D. Wheeler. *Secure Programming for Linux and Unix HOWTO*. Linux Documentation Project, 2003.
Online: <http://www.dwheeler.com/secure-programs/>.
- [33] M. Zalewski. Remote vulnerability in SSH daemon CRC32 compression attack detector, Feb. 2001.
Online: http://www.bindview.com/Support/RAZOR/Advisories/2001/adv_ssh1crc.cfm.