UCSD CSE 127: Introduction to Computer Security

Scribe Notes Lecture 2

These lecture notes were originally scribed by students in CSE 127 Winter 2021. They have been lightly edited but may contain errors.

Program Security

The topic of today's lecture is program security. There are two approaches to the question of when a program is secure: formal and pragmatic. In the *formal approach*, one says that a program is secure when it does exactly what it should. No more, no less.

This brings up another question: how would one know what a program is supposed to do? Someone could describe what the program is supposed to do, but would you trust them to accurately report it? Or do you write the code yourself?

The *pragmatic approach* would describe a program as secure when it does not do "bad" things. Oftentimes, it is easier to specify a list of "bad" things. Such examples include, but are not limited to: deleting or corrupting system files, crashing the system, divulging passwords over the internet, sending threatening emails to the professor, and more.

What if the program does not do bad things, but it could? Is it secure? It is complicated to define what this means, but it is mainly regarding the idea of intended functionality.

Weird Machines

Complex systems contain unintended functionality. More specifically, any systems that are complex enough are likely to contain some unintended functionality, like the system of government, for example.

The idea of a *weird machine* is different from a normal machine in that a normal machine has normal intended functionality, where some expected input will trigger it to output a normal thing; but on the other hand, a complex enough system will provide weird outputs when it receives unexpected inputs, which makes it a weird machine. Therefore, in the context of security, this unintended functionality can be triggered by the attackers.

Software Vulnerability

A *software vulnerability* is a bug in a program that allows unprivileged user capabilities that should be denied to them.

There are a lot of different kinds of vulnerabilities. One type of vulnerability are bugs that violate *control flow integrity*, which means this vulnerability allows an attacker to run code on your computer. A program has a control flow that governs how instructions are executed throughout the program. If an attacker takes control of the control flow, the attacker can change the behavior of the program. These vulnerabilities involve violating assumptions about the programming language of its runtime.

Exploiting Vulnerabilities

The *threat model* is victim code handling input that comes from across a security boundary. Some examples of this include:

- A program accepting a command-line argument or input string provided by the attacker.
- A computer login screen that accepts a username and password typed by the attacker.

- A program reading a file provided by an attacker.
- A network router or card handling a network packet constructed by the attacker.
- A web server application accepting HTTP requests from an attacker.
- An email server or email clients accepting, storing, and rendering email messages from an attacker.
- A PDF viewer displaying a PDF file provided by an attacker.
- A barcode scanner reading barcodes provided by the attacker.

The *security policy* is to protect integrity of execution and confidentiality of data from being compromised by any attacker, including malicious users or any other party which can access the system or program. The priority to protect execution and confidentiality of data, so it must be assumed that any interaction is done by a highly skilled and malicious user.

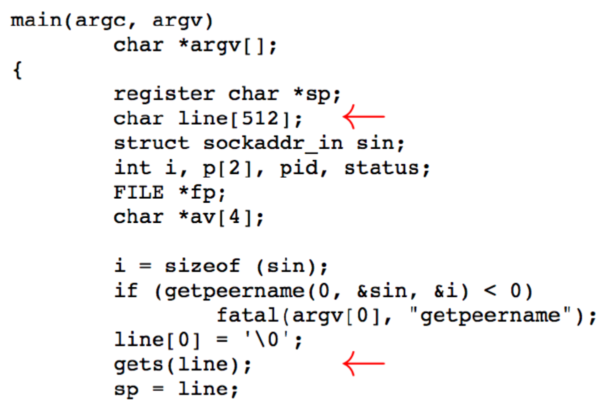
Buffer Overflows

Buffer overflows are an anomaly that occur when a program writes data beyond the boundary of a buffer. Overflowing a buffer can lead to attacks. Software written in C or C++ is particularly prone to buffer overflow vulnerabilities, as there is no bounds checking built into these languages. Examples of target software that have been found to be vulnerable to this attack in the wild include

- System software
- Web servers
- Browsers
- Operating Systems

When an attacker can provide inputs that a program processes using library functions with no bounds checking, such as `gets()`, `strcpy()`, or `strcat()`, the attacker may be able to cause the program to overwrite past the end of the target buffer with attacker-supplied values.

An example



```
main(argc, argv)
        char *argv[];
{
        register char *sp;
        char line[512];  ←
        struct sockaddr_in sin;
        int i, p[2], pid, status;
        FILE *fp;
        char *av[4];

        i = sizeof (sin);
        if (getpeername(0, &sin, &i) < 0)
                fatal(argv[0], "getpeername");
        line[0] = '\0';
        gets(line);      ←
        sp = line;
```

Figure 1: `fingerd` daemon in BSD 4.3

BSD 4.3 Finger Daemon: `finger` is a command used to get user information on multi-user Unix-based systems. This was more useful in the past when a single computer system typically served many users who logged in remotely. The finger command could also be used over the network: it would accept an argument, run the command locally, and report the output back. The code snippet above is from the `fingerd` daemon in BSD 4.3, which was released in 1986. In this implementation, the receiving server reads the remote request over the network, it allocates a 512 char buffer called `line` and reads into it using `gets()`. This works as intended when you input up to 511 chars (counting null terminator will result in 512 chars) but what happens if you type a 513 char string? It will overwrite after the buffer (2 bytes after exactly). Whatever exists next in memory will get overwritten. In this case, the buffer is on the stack so the next thing that was allocated in stack memory is overwritten.

Morris worm: In 1988, the Morris worm exploited the finger vulnerability using a buffer overflow attack to spread itself through the internet. The Morris worm was created by Robert Morris, a graduate student at Cornell who claimed to have been playing around with the finger vulnerability and ended up creating the Morris worm which got out of control. This lead to millions of dollars in damage, took down thousands of computers and had devastating effects on the internet. This resulted in the first conviction under the Computer Fraud and Abuse Act (CFAA).

You might think that this happened over 30 years ago, so surely buffer overflows are no longer a problem. However, Project Zero recently discovered a buffer overflow vulnerability inside the Qmage Remote codex in Samsung phones which allowed for unauthorized code execution on a Samsung phone.

How a buffer overflow lets you take over a machine

One of the properties exploited by buffer overflow and other attacks against program control flow vulnerabilities is that there is not necessarily a well-defined distinction between instructions and data in program execution. The conceptual boundaries and operation details can be different for different machine architectures.

As a program manipulates data, it changes values in memory to represent that data. The data may also influence the program execution. As an explicit example, a conditional branch will cause the program to execute some instructions if the data has one value and other instructions if the data has a different value. However, the instructions in a program are also just bits stored in memory that are interpreted by the CPU.

So in short, a program manipulates data, a program is data, and data manipulates a program flow. An attacker exploits these relationships in a buffer overflow attack.

What is a C array?

A C array is simply a pointer to a memory location in the stack or heap. The rest of the stack and heap are above and below it. An index into the array is simply an increment to the array pointer. The size of the array is used to allocate the specified amount of memory, but the language itself does not do any bounds checks when array values are read or written. Thus if a program writes past the end of the allocated array size, it will begin to overwrite other data in the stack or heap.

```
a[idx];    // This is equivalent
*(a+idx); // to this!
```

If you look at the C specification, it states that if we are accessing index *idx* inside array *a*, this is actually compiled to a pointer that points to the *a* plus the offset of *idx*. The specification says that these are identical. There is no language-based enforcement of a bound on the index.

Linux process memory layout

In Linux, each process has its own address space in memory. Inside this, it has everything it needs to actually execute the process. This includes all the library functions it might require, the stack (variables that are allocated by the code that is being run), the runtime heap, other data, and the actual instructions being executed by the program. In the diagrams in these notes, the stack is growing down while the heap is growing up.

As functions are called and return, stack frames are allocated and removed. So as the program is executed, the stack grows and shrinks.

The Stack

Note that `%` indicates a register in x86 assembly.

The stack pointer (`%esp`) will point to the current location of the top of the stack. The frame pointer or base pointer (`%ebp`) points to the caller's stack frame. Figure 3 shows a stack frame. When you call a function, a stack frame is set up by the code that is calling the function. The stack is divided into frames, and grows

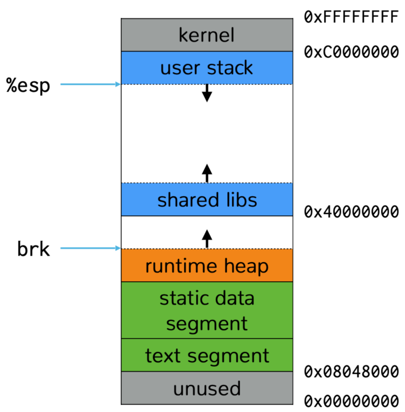


Figure 2: Linux Memory Layout

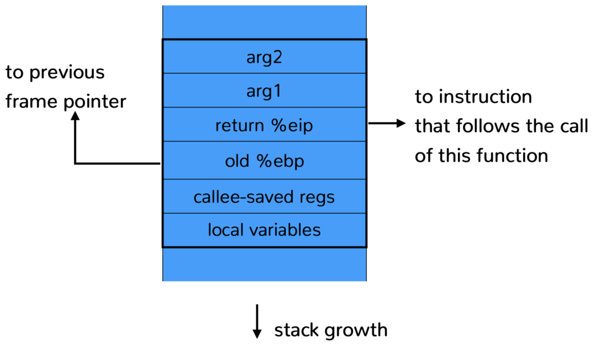

arg2
arg1
return %eip
old %ebp
callee-saved regs
local variables
to previous frame pointer
to instruction that follows the call of this function
stack growth

Figure 3: A Stack Frame

down (from high to low addresses). When you call a function, it updates the new stack pointer to point to the new top of the stack. Then you push on values onto the stack. You push on arguments to the function. Then you push on the location of the instruction pointer that should be returned to (return `%eip`) when this function returns. Then, you push on the location of the previous stack frame (old `%ebp`) so that this can be restored. This way you can continue the previous function after returning from the current function. After, you push in the saved registers from the callee so when the current function overrides various registers, the previous state can be restored when the current function returns. Then local variables are pushed onto the stack.

Brief review of x86 assembly

Everything that we do for this course in assembly will be in ATT syntax. Some of the main points are that registers are preceded by a % symbol and literal values are preceded by a $ symbol. When accessing memory, use the format `offset(memory-reference)` where the reference is the address and offset is the offset from the address.

Examples

- `movl %eax, %edx -> edx = eax`
  - This statement will assign the value of eax to edx
- `movl $0x123, %ecx -> edx = 0x123`
  - This statement will assign the value of 0x123 to edx
- `movl (%ebx), %edx -> edx = *((int32_t*) ebx)`
  - This statement will assign the value at the point in memory of ebx to edx.

- `movl 4(%ebx), %edx -> edx = *((int32_t*) ebx)`
  - This statement will assign the value at the point in memory of ebx + 4 to edx.

Brief Review of Stack Instructions

To push the register eax to the stack, that means moving the stack pointer down 4 bytes and assigning the value of `eax` to the stack pointer.

To pop we will do this in reverse order: assign value of `esp` to `eax` then add 4 to the stack pointer. In a 32-bit architecture, the 4 we are referring to is 4 characters.

When we call a specific instruction, what happens is we push the current instruction pointer to the stack and then move the instruction pointer to the instruction we are calling.

To return we pop the instruction pointer from the stack.

Leaving the function will need us to move the value of the base pointer into the stack pointer and then pop the value of the base pointer off the stack.

Demo 1 Overview

Some useful gdb commands:

- Launch a binary `foo` with gdb from terminal: `$ gdb foo`
- View disassembly output of a function `foo`: `(gdb) disas foo`
- Add a breakpoint `(gdb) b (line number, memory location)`

A program that asks for user input can be vulnerable to buffer overflow attacks. An input larger than the expected size but small enough to stay within the stack will be able to modify neighboring locations in memory.

Demo 1 showcases a simple overflow in this fashion that exploits `gets()`.

```
int main(int argc, char**argv) {
  char nice[] = "is nice.";
  char name[8];
  gets(name);
  printf(%s %s\n", name, nice);
  return 0;
}
```

In this demo, we exploit the functionality of `gets()`. When the input was less than 8 chars, the program printed the input and "is nice", then terminated, as expected. However, `gets()` will read as much input as we give to it, and when `printf()` tries to read from `name`, it will keep reading until it hits a null char. Therefore, when we make the input longer than 7 chars, we will overwrite into the `nice` buffer, and `printf()` will print out this extra data from the stack.

Demo 2 Overview

We can view the contents of the stack register in gdb using `x/32xw $esp`. `32xw` is a format string, meaning 32 hexadecimal words, starting at `$esp`, the stack register. This lets us see what's on the stack.

Demo 2 showcases a string buffer overflow exploit, namely

```
void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str)
}
```

where we control what `str` is. By making `str` longer than the buf, the string copy will continue writing into the stack. By knowing that above `buf`, the stack consists of the 4 bytes of `c`, 4 bytes of the ebp, and finally the 4 bytes of the saved return address, we can craft an input to overwrite the return address, controlling where the instruction pointer jumps to when exiting `func`.

This can be used to jump to existing code, or to attacker-provided shellcode.

Shellcode

Shellcode is a small code fragment that first gets control when an attack takes control of the instruction pointer (a *control flow hijack*). Early shellcode was used to execute a shell and use a separate exploit to gain root.

Shellcode can't have null characters (the string function will stop reading it) and if using `gets()`, also cannot contain newline chars (same reason).

The attacker might not know the address of where the shellcode is, so how can they know what to set the overwritten return address to? They can use a *NOP sled*: put `NOP` (No Operation) instructions in the string before the shellcode. If the control flow hijack returns anywhere within the buffer, it will execute NOPs until it hits the shellcode. This allows the return address to have errors: as long as it's in the NOP sled or directly on the shellcode, the shellcode will execute.