EECS 261: Computer Security **Fall 2007**

Lecture 4 — September 6

Lecturer: David Wagner *Scribe: DK Moon*

4.1 Required reading materials for this class

- Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns, Pincus and Baker
- Exploiting Format String Vulnerabilities
- Basic Integer Overflows

4.2 Reasons for vulnerabilities

- The lack of security design
- Architectural level flaws violating the design principles in Lecture 2
- Micro-level implementation bugs (The topic of this lecture)
- ...

4.3 Buffer Overrun

- *Morris worm* (1988): the first major Internet worm used the technique
- State of the art at that time: *stack smashing* only
- But, heap smashing is also possible

4.3.1 Stack Smashing

- By overwriting a return address stored on stack

```
f() {
    char buf[80], *p = buf;
    while ((*p = read_from_net()) != '\0') {
        ++p;
    }
}
```

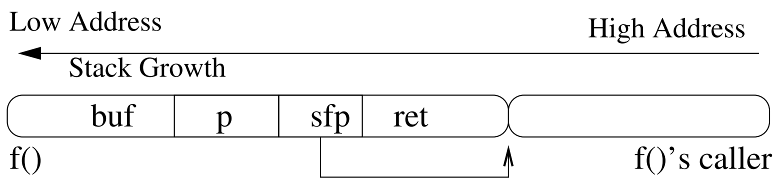

Figure 4.1. an example of *Stack smashing*. Stack grows from high address. A box with rounded corners represents a function frame.

The above code is vulnerable as there's no boundary check in line #3; it will write beyond *buf* unless null is given in the first 80 bytes. Figure 4.1 depicts the memory layout. An attacker can jump to wherever she wants by overflowing *buf* until overwriting the return address; when returning from *f()*, *CPU* will see the *modified return address*, pops the frame, and jumps to the address an attacker specified. In addition, an attacker can also put code into memory (i.e. in *buf*). So, an attacker can inject code and jump to the code.

4.3.2 Heap Smashing

For long time, people thought buffer overrun doesn't work for heap allocated buffer. However..

- At least, an attacker can crash a process by corrupting heap.
- An attacker can run arbitrary code she wants by keep overflowing heap-allocated buffer until overwriting a heap-allocated function pointer. This gives a control to an attacker not immediately, but after the overwritten function pointer is dereferenced. Figure 4.2 shows this scenario. Note that it's possible to calculate how many bytes to overflow if we can use some determinism: e.g. a vulnerable code is deterministic.

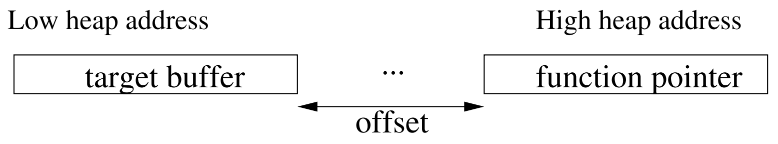

Figure 4.2. an example of *Heap smashing*. An attack can get a control by overwriting a function pointer on heap. She needs to overflow as many bytes as the offset; it's possible to calculate the offset when a process bases on determinism.

- A more sophisticated attack: an attacker can write any value to any location by overwriting a *heap control block* used by *malloc()* and *free()*; *malloc()* prepends a heap control block, like Figure 4.3, to maintain doubly-linked memory chunks. On freeing *header*, *free()* adjusts link pointers in a heap control block like this:

  $header \rightarrow next\_ptr \rightarrow prev\_ptr = header \rightarrow prev\_ptr$

Suppose that an attacker did overflow to fill some value, V, into *header* $\rightarrow$ *prev_ptr* and some address, A, into *header* $\rightarrow$ *next_ptr*. Now, calling *free()* results in $*(A+C) = V$, where C is the constant offset of *prev_ptr*. This lets an attacker write any value to arbitrary location. So, she can change return address, jump to malicious code, overwrite function pointer, and so on.

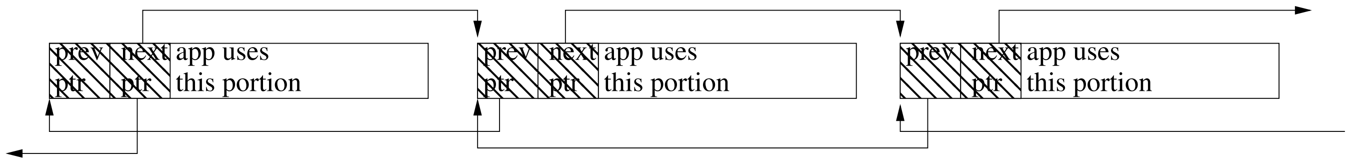

Figure 4.3. *malloc()* prepends *header control blocks*(shaded in the figure), which contain doubly-linked-list, to maintain memory chunks.

Why do we more worry about an attack for full control than a crash?

- Whole control gives more options (including crashes).
- Crashes are noticeable.
- Compromising a superuser-privileged process such as *sshd* means that a whole host is on an attacker's hand.

Buffer overrun is really common and easy to exploit. Saying like "it's very hard to exploit because an attacker should know every detail of program's behavior. So, it will give only limited control" is a totally wrong idea. There are tons of clever guys, and even a single-byte buffer overflow can cause significant impacts like changing frame pointer. We better assume all buffer overrun vulnerabilities exploitable and fix them right a way.

4.4 Format string vulnerabilities

```
1   f() {
2       char buf[512];
3       read(netfd, buf, sizeof(buf));
4       printf(buf); // buggy!
5   }
```

This code is buggy. The line #4 should have been *printf*("%s", *buf*). If a string supplied by an attacker contains *%d%d%d%d%d...*, *printf*() is going to look for arguments on stack, which don't exist there.

printf() has an internal pointer to look for format arguments on stack. For example, it expects 4-byte integer pointed by the pointer and increment the pointer by 4 bytes, when it encounters a format specifier %d in a format string. Figure 4.4 shows the call stack of the above code snippet. The internal pointer, p, reads and moves beyond $printf()$'s stack frame because there is no corresponding format argument on the stack. Therefore, we can move the pointer to the stack location storing $f()$'s return address by multiple '%d's. (something like '%100d' can be used to reduce many %d's.) And, we can also read the adress value by %x.

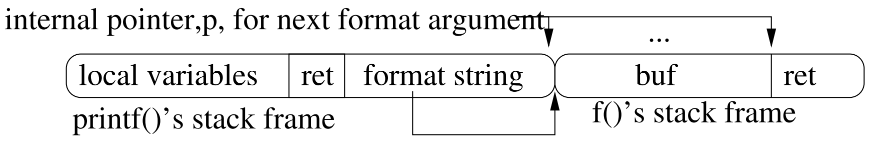

Figure 4.4. *printf()* has an internal pointer to seek for format arguments. We can move the pointer toward $f()$'s *ret* by using format specifiers

Worse, an attacker can also write. And she can specify both an address and a value to write!

- *Specifying an address*: An attacker can also write by '%n'. Basically, '%n' writes the number of printed characters so far into a given address. An attacker can supply a target address in buf, then %n touches the address.
- *Specifying a value*: A returned value by %n is usually small as we do not write such many characters. So, 4-byte value is created by concatenating four 1-byte values. For example, we can write 4-byte value into A by using four '%n's with overlapping addresses $(A+0)$, $(A+1)$, $(A+2)$, and $(A+3)$ (works for little endian machines).

4.5 Double free vulnerability

This causes heap corruption by calling $free()$ twice with the same argument. Either a crash or taking over a control.

4.6 Integer overflow

It occurs when down-casting or up-casting between integer variables of different sizes. This can cause buffer overrun when overflowed or underflowed integer is used as a buffer index.

Then, How to prevent integer overflows?

- Checking integer arithmetics every time (e.g. C#)
- Not allowing implicit type castings

- Using big integers: (e.g LISP, SmallTalk, and so on): Downside: performance trade-off; malicious user can pick numbers to hurt performance
- Supporting range types and making sure all arithmetic results always reside in the range (e.g. int [0-512])

4.7 Memory Safety

Vulnerabilities so far violates the *memory safety*: writing outside the range of an object, violating the stack integrity, violating the integrity of a control flow, ... Violations of memory safety are pretty sure indications of security holes.

Memory-safe languages prevent these violations by defining language semantics. (e.g. all array should be bounce-checked) All strongly type-safe languages are memory safe. For example, java, C#, LISP, Scheme, ML, etc.

So, we can use a memory-safe language when writing a security critical program. Then, why still C is widely used?

- low-level control. e.g. memory mapped i/o
- performance
- many legacy programs and libs are in C
- everybody knows C
- easy integration: a lot of system interfaces are in c/c++

4.8 Command Injection Vulnerabilities

It usually pops up when control and data interleave in the same channel. For example, format string vulnerabilities are due to mixing control and data; we expect data from user, but an attacker gives a control. Type checking doesn't help here. So, even memory-safe languages are under risk, too.

- Shell meta character injection:e.g. "*system("mail " + $addr)*": a user can use semicolon
- SQL injection: "*select from users where name = $username*" : *username* can be another SQL query. *username* can contain a semicolon.

How can we prevent these command injection vulnerabilities?

- sanitization by the whitelist of characters
- do not use string, but use object type (e.g. passing SQL parse tree rather than using string concatenation or something

$\longrightarrow$ **Lesson from pay-phone story**: separate control and data channel!