Foundations of Network and Computer Security

John Black

Lecture #20
Nov 4th 2004

CSCI 6268/TLEN 5831, Fall 2004

Announcements

- Quiz #3 – Today
  - Need to know what big-endian is
  - Remind me to mention it if I forget!
- Brian's detective work
  - Mazdak will fix as he can

The Translation Table

Original C Source	Derived Abstract Model
`char s[n];`	`alloc(s) = n;`
`s[n] = '\0';`	`len(s) = max(len(s), n+1)`
`p = "foo";`	`len(p) = 4; alloc(p) = 4;`
`strlen(s)`	`len(s)-1`
`gets(s);`	`len(s) = choose(1…∞);`
`fgets(s,n,…);`	`len(s) = choose(1…n);`
`strcpy(dst, src);`	`len(dst) = len(src);`
`strncpy(dst, src, n);`	`len(dst) = min(len(src), n);`
`strcat(s, suffix);`	`len(s) += len(suffix) – 1;`
`strncat(s, suffix, n);`	`len(s) += min(len(suffix)–1,n);`
`And so on . . .`	

Program Analysis

- Once we set these “variables” we wish to see if it’s possible to violate our constraint (len(s) <= alloc(s) for all strings s)
  - A simplified approach is to do so *without* flow analysis
    - This makes the tool more scalable because flow analysis is hard
    - However it means that **`strcat`**() cannot be correctly analyzed
    - So we will flag every nontrivial usage of **`strcat`**() as a potential overflow problem (how annoying)
- The actual analysis is done with an “integer range analysis” program which we won’t describe here
  - Integer range analysis will examine the constraints we generated above and determine the possible ranges each variable could assume

Evaluating the Range Analysis

- Suppose the range analysis tells us that for string s we have

  a <= len(s) <= b and c <= alloc(s) <= d

- Then we have three possibilities:

  b <= c s never overflows its buffer

  a > d s always overflows its buffer (usually caught early on)

  c <= b s possibly overflows its buffer: issue a warning

An Implementation of the Tool

- David Wagner implemented (a simple version of) this tool as part of his PhD thesis work
  - Pointers were ignored
    - This means `*argv[]` is not handled (and it is a reasonably-frequent culprit for overflows)
  - `structs` were handled
    - Wagner ignored them initially, but this turned out to be bad
  - function pointers, `unions`, ignored

Emperical Results

- Applied to several large software packages
  - Some had no known buffer overflow vulnerabilities and other did
- The Linux **`nettools`** package
  - Contains utilities such as **`netstat`**, **`ifconfig`**, **`route`**, etc.
  - Approximately 7k lines of C code
  - Already hand-audited in 1996 (after several overflows were discovered)
  - Nonetheless, Wagner discovered several more exploitable vulnerabilities

And then there's **`sendmail`**

- **`sendmail`** is a Unix program for forwarding email
  - About 32k lines of C
  - Has undergone several hand audits after many vulnerabilities were found (overflows and race conditions mostly)
  - Wagner found one additional off-by-one error
- Running on an old version of **`sendmail`** (v. 8.7.5), he found 8 more (all of which had been subsequently fixed)

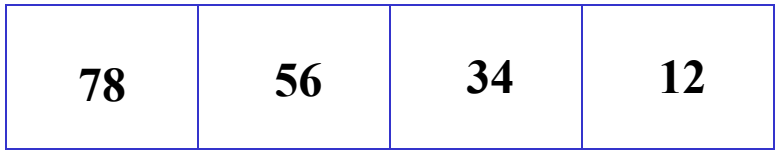
Performance

- Running the tool on **`sendmail`** took about 15 mins
  - Almost all of this was for the constraint generation
  - Combing by hand through the 44 “probable overflows” took much longer (40 of these were false alarms)
  - But **`sendmail`** 8.9.3 has 695 calls to potentially unsafe string routines
    - Checking these by hand would be 15 times more work than using the tool, so running the tool *is* worthwhile here

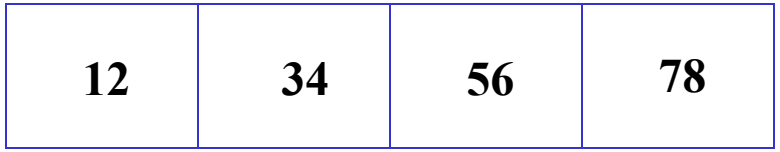
Endianness

- A multi-byte quantity (like an integer) can be stored in two ways
  - i = 0x12345678;
  - In memory:

Lower Addrs



Little Endian



Big Endian

Off-by-one Overflows

- Consider this code:

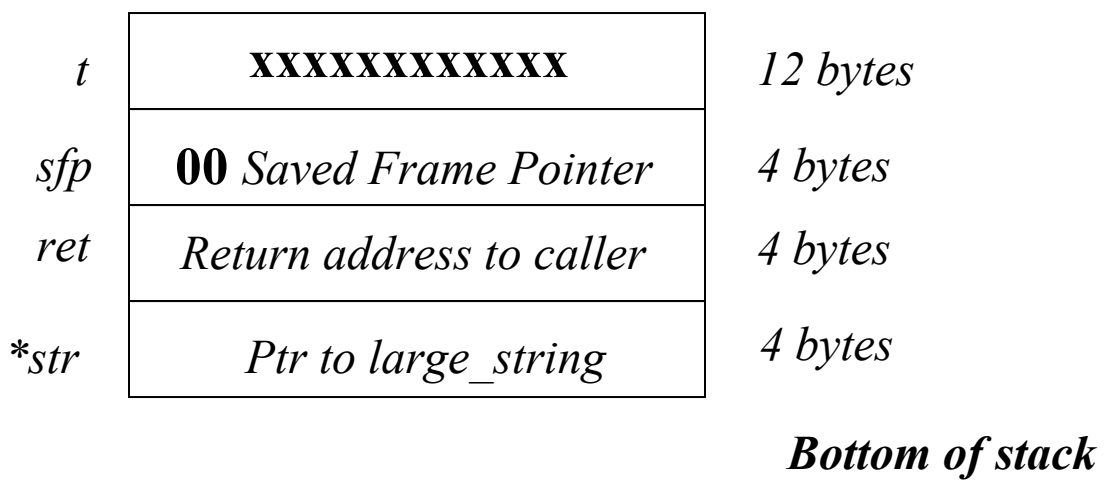
```
void test1(char *p)
  {
     char t[12];
    strcpy(t, "test");
    strncat(t, p, 12-4);
  }
```

- Recall that strncat() adds chars from p on to string t, adding at most 12-4=8 of them
  - But with the null, this produces an off-by-one error: we need 13 characters!
  - Note: this is a common error and usually not thought of as a security problem!

What happens on overflow?

- Try test1("xxxxxxxx")
  - Null byte overwrites first byte of sfp
  - Next we mov esp, ebp; pop ebp
    - This means the ebp will contain the old value clobbered by the Null (call this mbp: munged base pointer)

Note: off-by-one must be adjacent to sfp in order to be exploitable



And on the next function exit?

- With the wrong ebp value, equal to mbp, we return to the caller
  - Caller then exits next and does what?
  - mov esp, ebp; pop ebp; ret
  - So the stack ptr is now mbp
  - If we also control memory around address mbp, we take over the machine
    - The ret call will transfer control wherever we like
    - Note that we don't need an overflow in this secondary buffer, we just need to control its contents
  - Despite this sounding far-fetched, there have been numerous exploitable off-by-ones
    - SSH, wu-ftp, and more

Format String Vulnerabilities

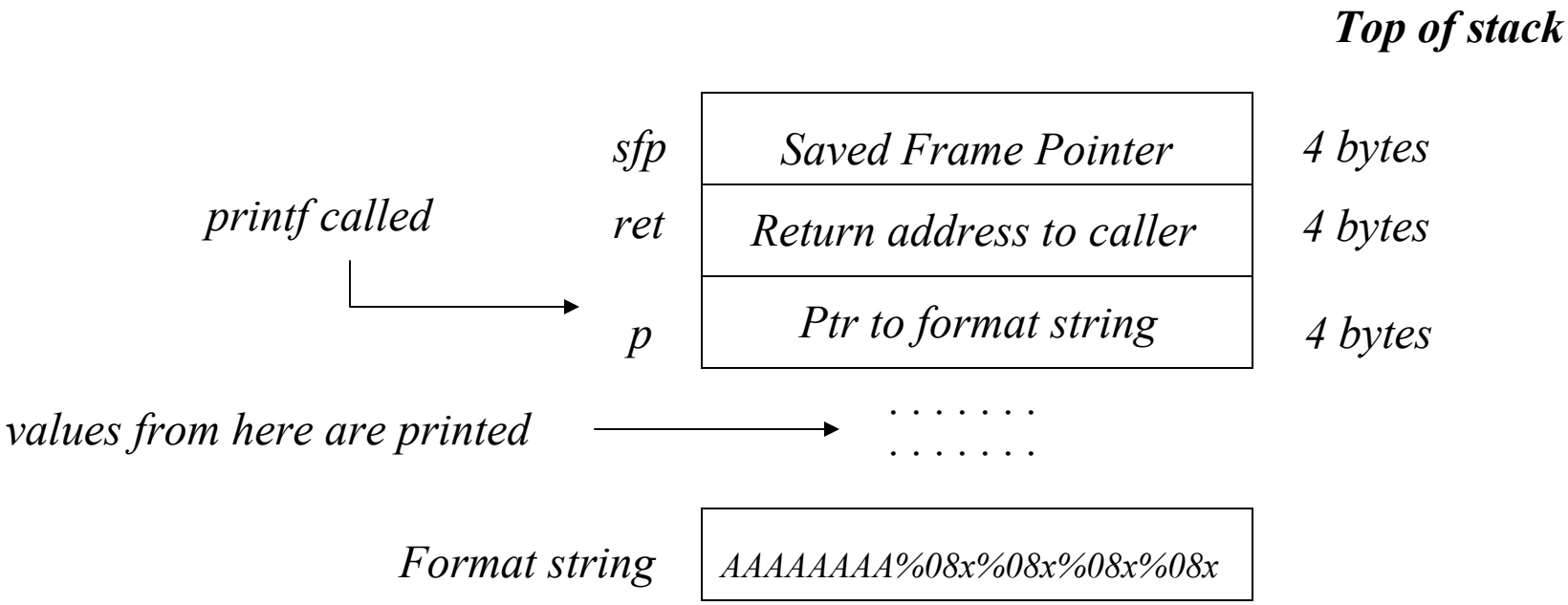
- Example:

```
output(char *p)
{
    printf(p);
}
```

- Seems harmless: prints whatever string is handed to it
  - But if string is user-supplied, strange things can happen
  - Consider what happens if formatting chacters are included
    - Ex: p = “%s”

Format Strings (cont)

- Let's play with format strings:
  - "AAAAAAA%08x%08x%08x%08x"
  - Prints values from the stack (expecting parameters)



Example Output

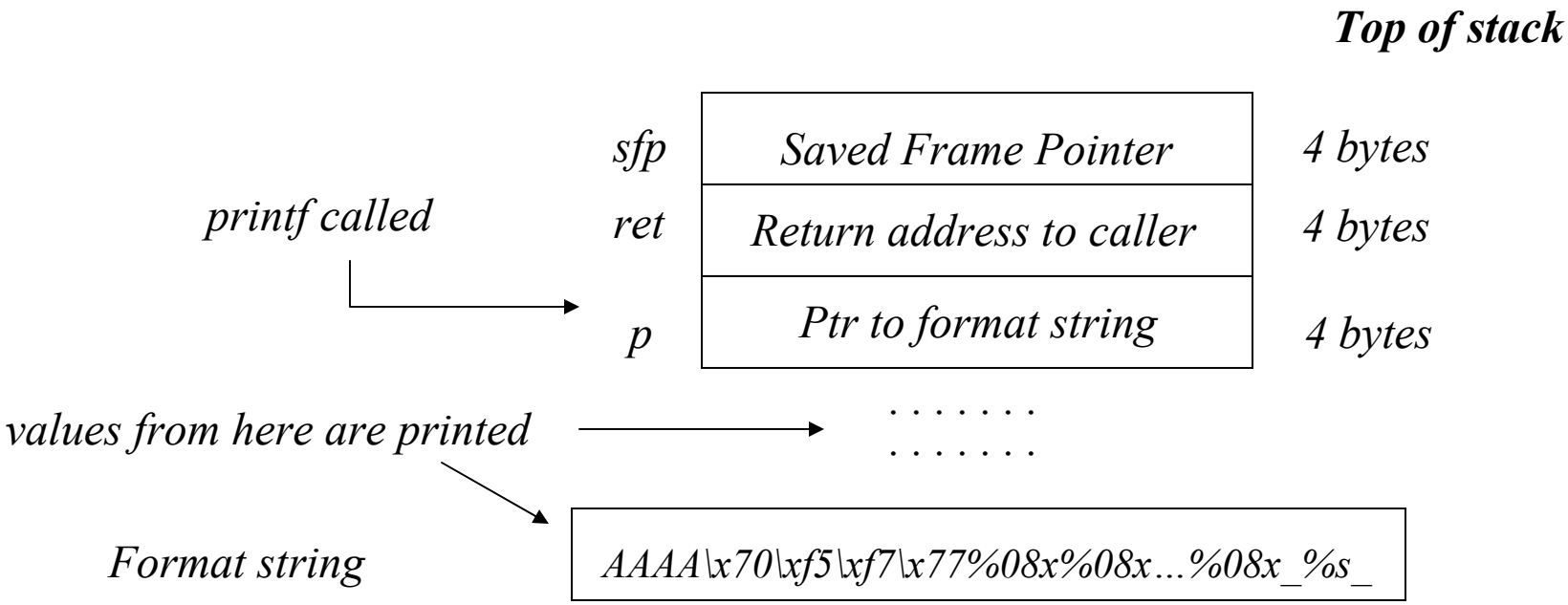
- Continuing with "AAAAAAA%08x%08x%08x%08x"
  - AAAAAAAA012f2f1580ff000010ff202018ae1414
  - So the above values were on the stack… how can we exploit this?
    - We can keep printing stack values until we run into the format string itself… might lead to something interesting
    - AAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
    - Output: AAAAAAAA12f2f1580f…41414141414141414178380425

Printing Data from (almost) Anywhere in Memory

- As we saw, %s interprets stack value as a pointer, not an int
  - Suppose we would like to read from address 0x77f7f570
    - Note: we can't have any 00 bytes in the address since we are about to embed it in a string
  - Use format string
    "AAAA\x70\xf5\xf7\x77%08x%08x…%08x_%s_"
    - Note we're assuming little-endian here
  - Output "AAAApJ^0012ff800cccc…ccc41414141_&h2!$*\&_"
    - Note that string will terminate at first 0 byte encountered (and segfault if you go off the end of valid memory)

Picture of Stack

- Kind of confusing:
  - As printf reads the format string, it's reading down the stack for its arguments as well
  - When printf gets to the %s, the arg ptr is pointing at \x70\xf5\xf7\x77, so we print the contents of that addr



But Can We *Alter* the Stack Contents?

- Introducing the %n token
  - This one is obscure: nothing is printed but the number of chars printed thus far is stored at the address indicated by the corresponding parameter to %n
  - Ex: printf("hi%n there", &i); now i = 2
  - How can we use this ability to write to memory?
    - Consider "AAAA\x70\xf5\xf7\x77%08x%08x…%08%n"
    - Writes 0x00000164 (= 356) to address 0x77f7f570

Using %n

- Extending this, we can write any value of our choice to (almost) any address
  - "AAAA**\x70**\xf5\xf7\x77**\x71**\xf5\xf7\x77**\x72**\xf5\xf7\x77**\x73**\xf5\xf7\x77%08x%08x…%08x%n%n%n%n"
  - Writes 0x00000164 four times, so at address 0x77f7f570 we will see 0x64646464
  - But how do we get values of our choice to address 0x77f7f570 instead of this 0x64646464 thing?
    - Let's use the %##u token (or any other that takes a length specifier)

Writing Arbitrary Values

- We use the width specifier to add any number of bytes we like to the current "number of printed chars" count
  - To write 0xfff09064 we use ""AAAA**\x70**\xf5\xf7\x77**\x71**\xf5\xf7\x77**\x72**\xf5\xf7\x77**\x73**\xf5\xf7\x77%08x%08x…%08x%n%43u%n%96%n%15u%n"
  - This works fine if we are wanting to write ever-increasing byte values
    - How can we write 0xf0ff9064?
    - How might we write to address 0x400014a0?

Detecting Format String Vulnerabilities

- Not as hard to detect as buffer overflows (which can be *very* subtle)
- One method is to look for calls to printf, sprintf, snprintf, fprintf, etc. and examine the stack clean up code
  - Recall that after a function call returns, it must remove its parameters from the stack by adding the sum of their sizes to esp
  - If we see **`add $4, %esp`**, we flag a possible vulnerability

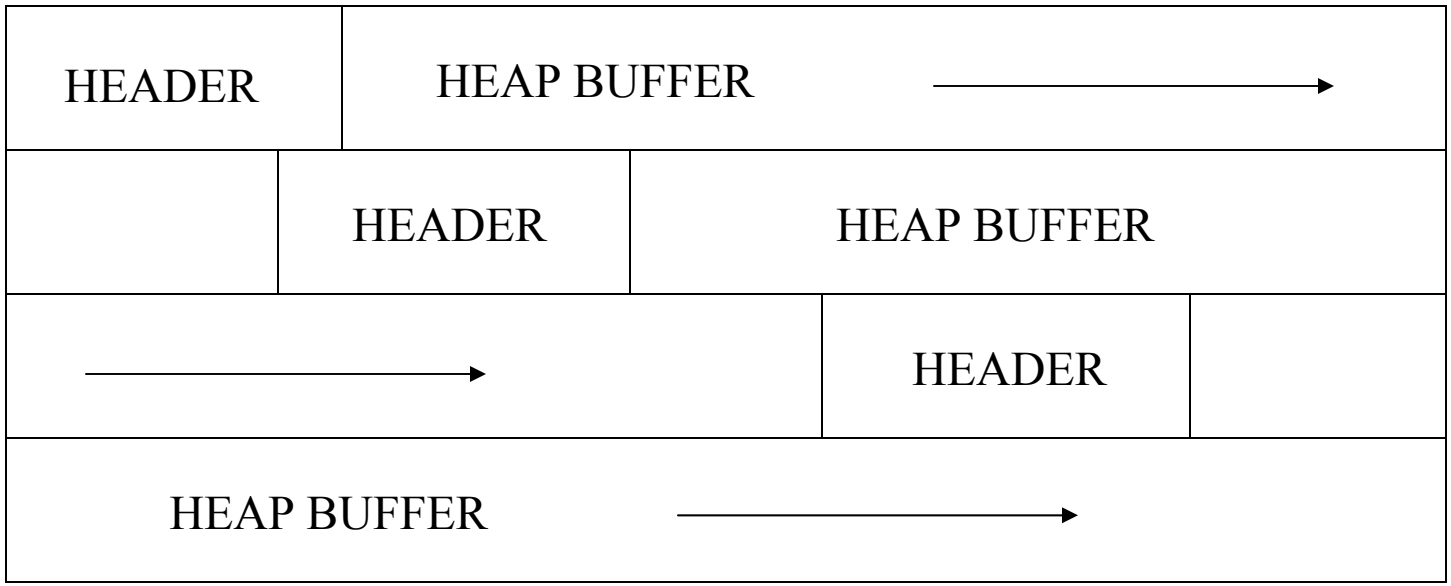
Heap Overflows

- These are among the hardest to exploit and depend on minute OS and compiler details
  - Some hackers consider writing a heap overflow as a rite of passage
  - We will only sketch how they work; a detailed example would take too long
  - This is the last software vulnerability we'll talk about in this class, but there are MANY more

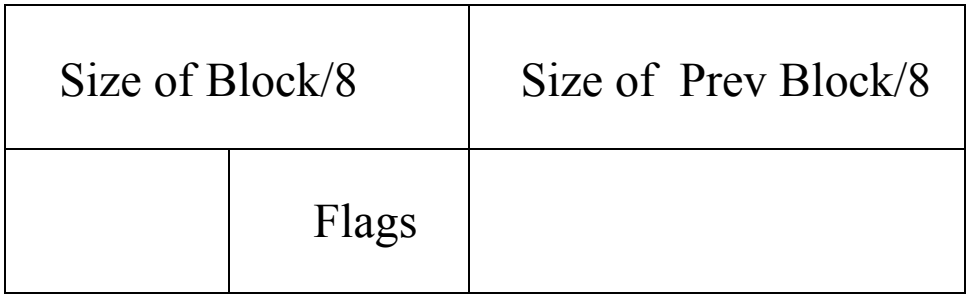
What is the Heap?

- The area of data which grows toward the stack
  - malloc() and new use this memory area for dynamic structures
  - Unlike the stack, we do not linearly grow and shrink the heap
    - We allocated and deallocate blocks in any order
    - We have to worry about marking the size of blocks, blending adjacent deallocated chunks for re-use, etc.
    - Many algorithms (with various tradeoffs) exist so this attack will depend on the specifics of those algorithms

The Heap (Layout)



Higher Memory



Windows 2K
Heap Header

How to Exploit a Heap Overflow

- Details vary, but in one case:
  - free() takes a value from the header and writes to an address also taken from the header
  - If we can overflow the buffer just before this header, we can control both the address used and the value written to that address
  - This address could be a return address on the stack, and we know the rest of the story…