Foundations of Network and Computer Security

John Black

Lecture #21
Nov 9th 2004

CSCI 6268/TLEN 5831, Fall 2004

Announcements

- Quiz #3 – Returned today
- Proj #2 – Due week from Thurs
- Proj #3 – Still time, but get started
  - Tricky in parts
- Use of class mailing lists
  - Good!

Format String Vulnerabilities

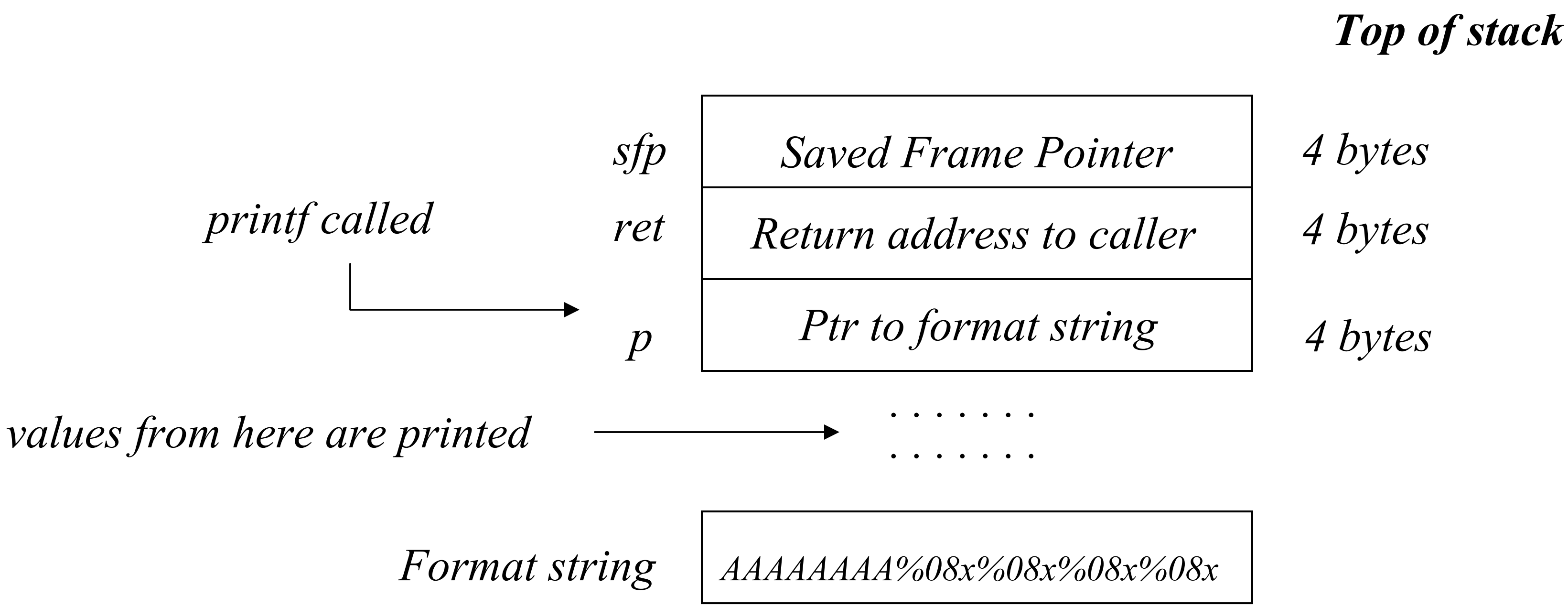
- Example:

```
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

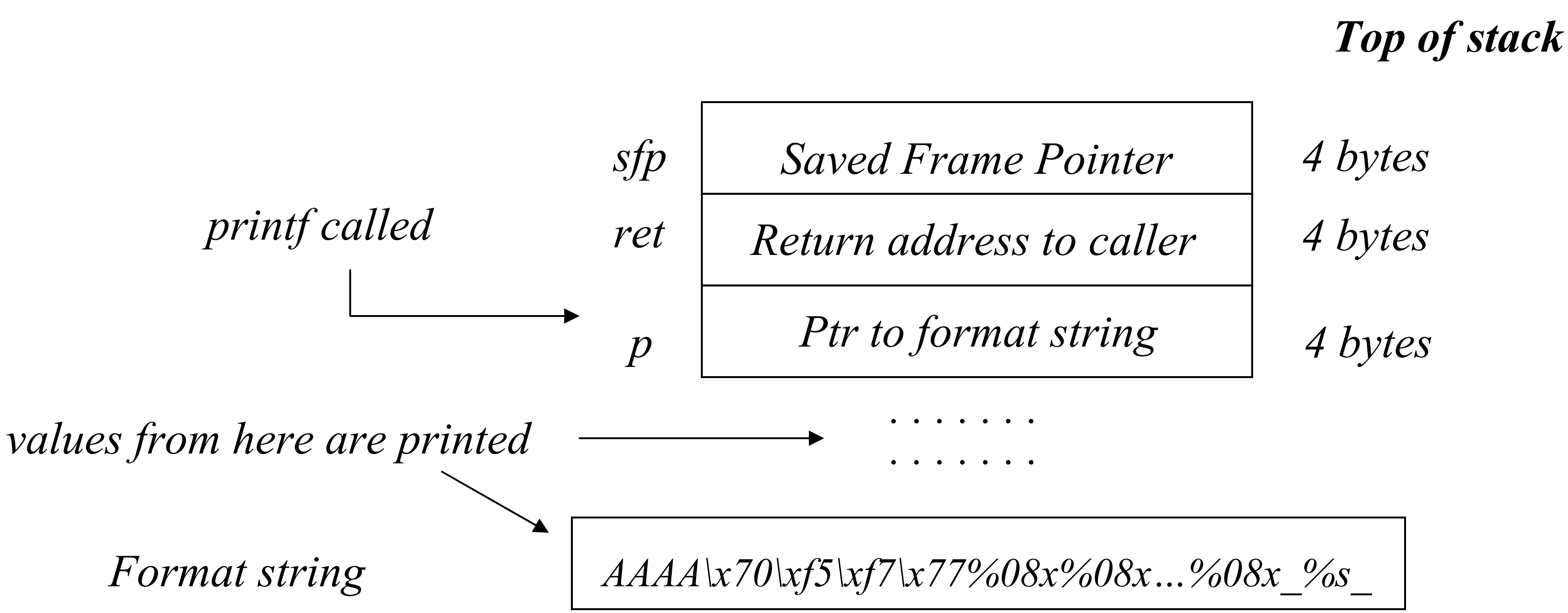
- Continuing with "AAAAAAA%08x%08x%08x%08x"
  - AAAAAAAA012f2f1580ff000010ff202018ae1414
  - So the above values were on the stack… how can we exploit this?
    - We can keep printing stack values until we run into the format string itself… might lead to something interesting
    - AAAAAAA%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%0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

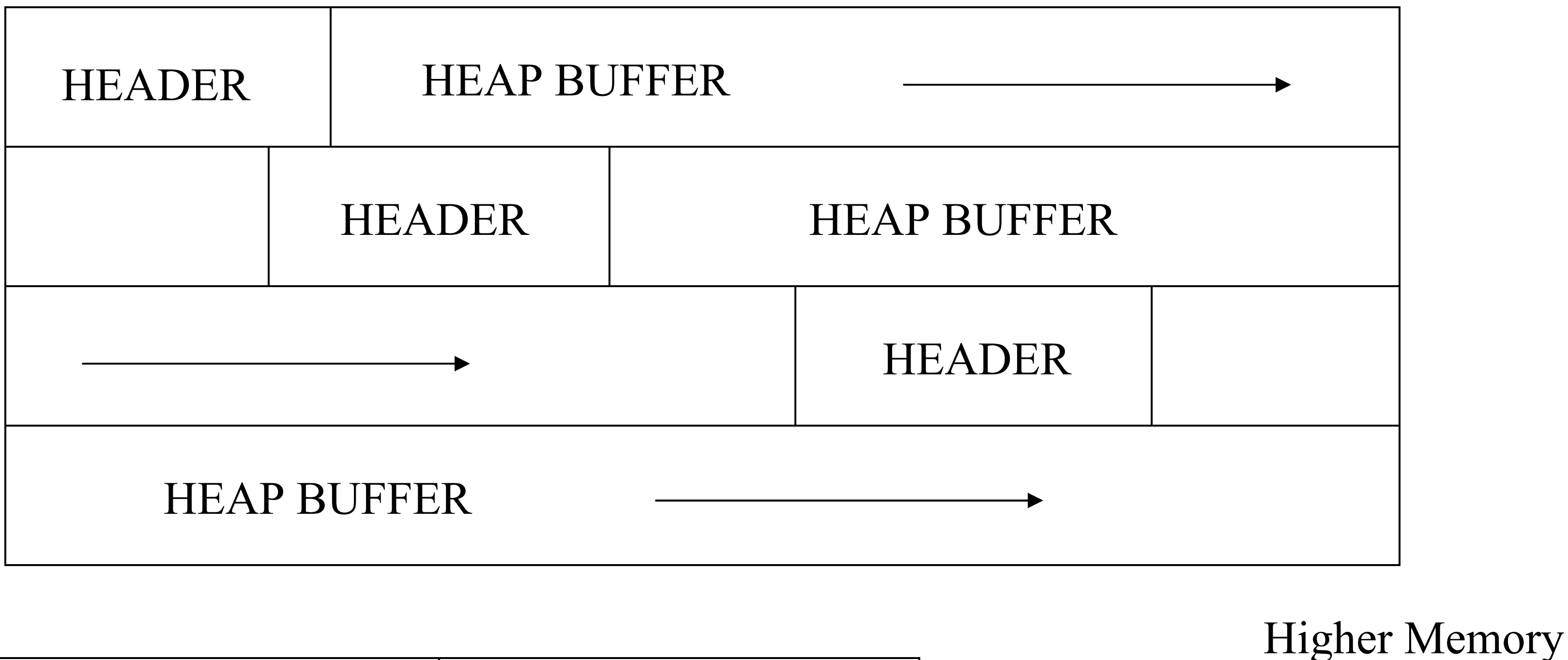
Heap Overflows

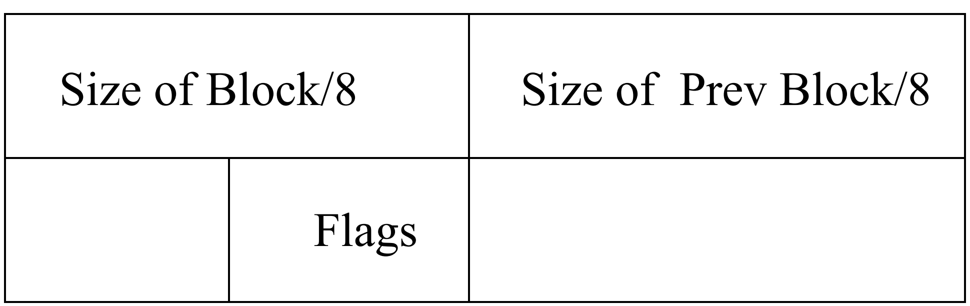
- These are among the hardest to exploit and depend on minute OS and compiler details
  - Some hackers consider writing a heap overflow as a rite of passage
  - We will only sketch how they work; a detailed example would take too long
  - This is the last software vulnerability we'll talk about in this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





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…

Other Vulnerabilities

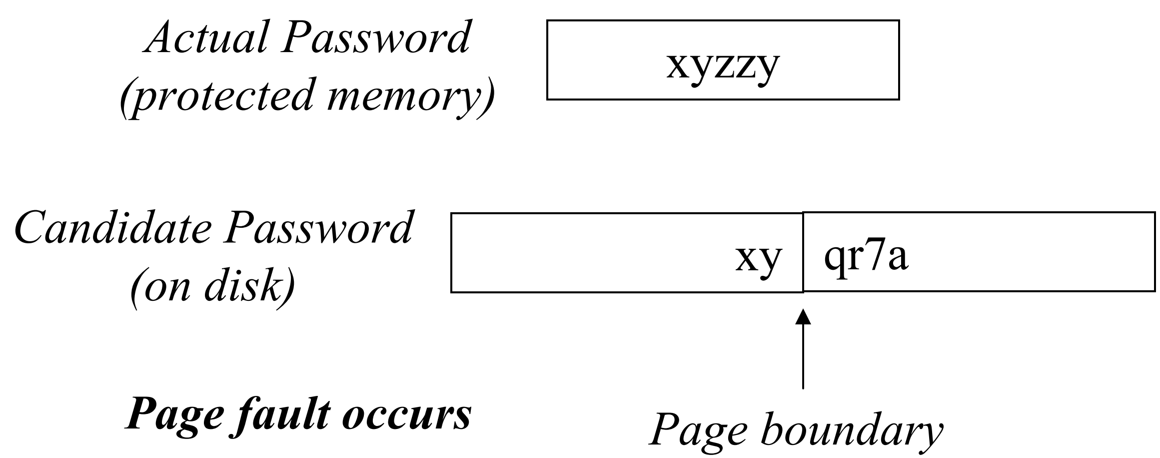
- We have been discussing a range of common and generic vulnerabilities
  - There are lots more which are more application-specific
  - We couldn’t possibly hope to cover them all
  - Let’s look at a couple of examples

Password Checking and Page Faults

- Some older OS worked like this:
  - Password was checked character-by-character by a high-privilege program
  - If password mismatch occurred, program stopped checking at that point
  - Page faults were viewable by all
  - Idea:
    - Put candidate password on disk which is known not to be in memory, and watch page faults

Page Fault Technique (cont)

- Idea: place candidate password across page boundary on disk
  - If we page fault to get second page, the password-checking program must have matched correctly up to all characters before the boundary
  - If we don't page fault, keep trying last letter before boundary
  - Each time we get a character correct, shift left and continue until we get the whole password



Password Crackers

- Unix approach: store one-way hash of password in a public file
  - Since hash is one-way, there is no risk in showing the digest, right?
  - This assumes there are enough inputs to make exhaustive search impossible (recall IP example from the midterm)
  - There are enough 10-char passwords, but they are NOT equally likely to be used
    - HelloThere is more likely than H7%$$a3#.4 because we're human

Password Crackers (cont)

- Idea is simple: try hashing all common words and scan for matching digest
  - Original Unix algorithm for hash is to iterate DES 25 times using the password to derive the DES key
    - $DES^{25}(pass, 0^{64})$ = digest
    - Note: this was proved secure by noticing that this is the CBCMAC of $(0^{64})^{25}$ under key 'pass' and then appealing to known CBCMAC results
    - Why is DES iterated so many times?

Password Crackers (cont)

- Note: Actually uses a variant of DES to defeat hardware-based approaches
- Note: Modern implementations often use md5 instead of this DES-based hash
- But we can still launch a ‘dictionary attack’
  - Take large list of words, names, birthdays, and variants and hash them
  - If your password is in this list, it will be cracked

Password Crackers: example

word	*digest*
alabaster	xf5yh@ae1
albacore	&trh23Gfhad
alkaline	Hj68aan4%41
wont4get	7%^^1j2labdGH

Pasword file * /etc/passwd*
jones:72hadGKHHA%
smith:HWjh234h*@!!j!
jackl:UwuhWuhf12132^
taylor:Hj68aan4%41
bradt:&sdf29jhabdjajK22
knuth:ih*22882h*F@*8haa
wirth:8w92h28fh*(Hh98H
rivest:&shsdg&&hsgDGH2

Making Things Harder: Salt

- In reality, Unix systems always add a two-character “salt” before hashing your password
  - There are 4096 possible salts
  - One is randomly chosen, appended to your password, then the whole thing is hashed
  - Password file contains the digest and the salt (in the clear)
  - This prevents attacking all passwords in /etc/passwd in parallel

Password Crackers: with Salt

Table for Salt Value: ***A6***

word	*digest*
alabaster	xf5yh@ae1
albacore	&trh23Gfhad
alkaline	U8&@H**12
wont4get	7%^^1j2labdGH

no match

Pasword file /etc/passwd
jones:72hadGKHHA%**H7**
smith:HWjh234h*@!!j!**YY**
jackl:UwuhWuhf12132^**a$**
taylor:Hj68aan4%41**y$**
bradt:&sdf29jhabdjajK22**Ja**
knuth:ih*22882h*F@*8haa**U%**
wirth:8w92h28fh*(Hh98H**1&**
rivest:&shsdg&&hsgDGH2***1**

Fighting the Salt: 4096 Tables

- Crackers build 4096 tables, one for each salt value
  - Build massive databases, on-line, for each salt
    - 100’s of GB was a lot of storage a few years ago, but not any longer!
    - Indexed for fast look-up
    - Most any common password is found quickly by such a program
    - Used by miscreants, but also by sysadmins to find weak passwords on their system

Getting the /etc/passwd File

- Public file, but only if you have an acct
  - There have been tricks for remotely fetching the /etc/passwd file using ftp and other vulnerabilities
  - Often this is all an attacker is after
    - Very likely to find weak passwords and get on the machine
  - Of course if you are a *local* user, no problem
  - Removing the /etc/passwd from global view creates too many problems

Shadowed Passwords

- One common approach is to put just the password digests into /etc/shadow
  - /etc/passwd still has username, userid, groupid, home dir, shell, etc., but the digests are missing
  - /etc/shadow has only the username and digests (and a couple of other things)
  - /etc/shadow is readable and writeable for root only
    - Makes it a bit harder to get a hold of
    - Breaks some software (including the buggy web server) which wants to authenticate users with their passwords
      - One might argue that non-root software shouldn't be asking for user passwords anyhow

Last Example: Ingres Authorization Strings

- Ingres, 1990
  - 2nd largest database company behind Oracle
- Authorization Strings
  - Encoded what products and privileges the user had purchased
    - Easier to maintain this way: ship entire product
    - Easier to sell upgrades: just change the string
- Documentation guys
  - Needed an example auth string for the manual

Moral

- There’s no defending against stupidity

- Social engineering is almost always the easiest way to break in
  - Doesn’t work on savvy types or sys admins, but VERY effective on the common user
  - I can almost guarantee I could get the password of most CU students easily
    - “Hi this is Jack Stevens from ITS and we need to change your password for security reasons; can you give me your current password?”

Social Engineering: Phishing

- Sending authentic looking email saying “need you to confirm your PayPal account information”
  - Email looks authentic
  - URL is often disguised
  - Rolling over the link might even pop-up a valid URL in a yellow box!
  - Clicking takes you to attacker’s site, however
    - This site wants your login info

Disguising URLs

- URI spec
  - Anything@http://www.colorado.edu is supposed to send you to www.colorado.edu
    - Can be used to disguise a URL:
      - ***http://www.ebay.com-SECURITYCHECKw8grHGAkdj>jd7788<Account Maintenace-4957725-s5982ut-aw-ebayconfirm-secure-23985225howf8shfMHHIUBd889yK@www.evil.org***
    - Notice feel-good words
    - Length of URI exceeds width of browser, so you may not see the end
    - www.evil.org could be hex encoded for more deception

Disguising URL’s (cont)

- This no longer works on IE
- Still works on Mozilla
- In IE 5.x and older, there was another trick where you could get the toolbar *and* URL window to show “www.paypal.com” even though you had been sent to a different site
  - Very scary
- Moral: don’t click on email links; type in URL manually