

CS 4803

Computer and Network Security

Alexandra (Sasha) Boldyreva
Application-level security. Buffer overflows.
Malicious code. Worm and viruses.

1

Application-level security

- I.e., programming-language security
- Previous focus was on protocols and algorithms to prevent attacks
 - Are they implemented correctly?
- Here, focus is on programming errors and how to deal with them
 - Reducing/eliminating/finding errors
 - Containing damage resulting from errors

2

Classifying flaws

- Intentional flaws
 - E.g., “backdoors”
- Unintentional flaws
 - E.g., programmer errors

3

Buffer overflows

- 50% of reported vulnerabilities
- Overflowing a buffer results in data written elsewhere:
 - User’s data space/program area
 - System data/program code
 - Including the stack, or memory heap

4

Stack Buffers

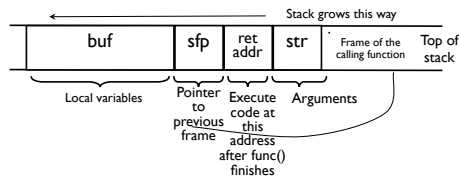
- Suppose Web server contains this function

```
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack



5

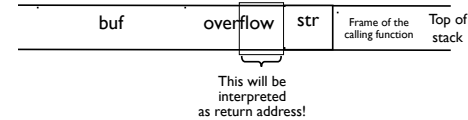
What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...

```
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

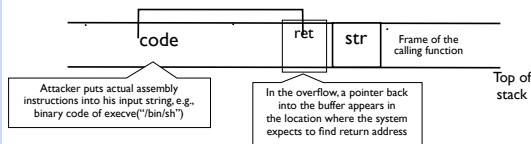
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



6

Executing Attack Code

- Suppose buffer contains attacker-created string



- When function exits, code in the buffer will be executed

7

Problem: No Range Checking

- strcpy does not check input size
- strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe

8

Off-By-One Overflow

Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
    char buffer[512]; int i;

    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}

void main(int argc, char *argv[]) {
    if (argc==2)
        notSoSafeCopy(argv[1]);
}
```

This will copy 513 characters into buffer. Oops!

1-byte overflow: can't change RET, but can change pointer to previous stack frame

9

Finding buffer overflows

- Hackers find buffer overflows as follows:
 - Run web server on local machine
 - Issue requests with long tags. All long tags end with "\$\$\$\$"
 - If web server crashes: search core dump for "\$\$\$\$" to find overflow location.

10

Addressing buffer overflows

- Basic stack exploit can be prevented by marking stack segment as non-executable, or randomizing stack location.
 - Code patches exist for Linux and Solaris.
- Problems:
 - Some apps need executable stack (e.g. LISP interpreters).
 - Does not block more general overflow exploits
- Patch not shipped by default for Linux and Solaris

11

Run-time checking: StackGuard

- Embed "canaries" in stack frames and verify their integrity prior to function return



12

Run-time checking: Libsafe

- Intercepts calls to strcpy (dest, src)
 - Validates sufficient space in current stack frame
 - If enough space, does strcpy. Otherwise, terminates application

13

More methods ...

- Address obfuscation
 - Encrypt return address on stack by XORing with random string. Decrypt just before returning from function.
 - Attacker needs decryption key to set return address to desired value.

14

Preventing Buffer Overflow

- Use safe programming languages, e.g., Java
 - What about legacy C code?
- Mark stack as non-executable
- Make buffers (slightly) longer than necessary to avoid "off-by-one" errors
- Randomize stack location or encrypt return address on stack by XORing with random string
 - Attacker won't know what address to use in his string
- Static analysis of source code to find overflows
- Run-time checking of array and buffer bounds
 - StackGuard, libsafe, many other tools
- Black-box testing with long strings

15

Viruses/malicious code

16

Viruses/malicious code

- Virus – passes malicious code to other non-malicious programs
 - Or documents with “executable” components
- Trojan horse – software with unintended side effects
- Worm – propagates via network
 - Typically stand-alone software, in contrast to viruses which are attached to other programs

17

Viruses

- Can insert themselves before program, can surround program, or can be interspersed throughout program
 - In the last case, virus writer needs to know about the specifics of the other program
- Two ways to “insert” virus:
 - Insert virus in memory at (old) location of original program
 - Change pointer structure...

18

Viruses...

- Boot sector viruses
 - If a virus is loaded early in the boot process, can be very difficult (impossible?) to detect
- Memory-resident viruses
 - Note that virus might complicate its own detection
 - E.g., removing virus name from list of active programs, or list of files on disk

19

Some examples

- BRAIN virus
 - Locates itself in upper memory; resets the upper memory bound below itself
 - Traps “disk reads” so that it can handle any requests to read from the boot sector
 - Not inherently malicious, although some variants were

20

Morris worm (1988)

- Resource exhaustion (unintended)
 - Was supposed to have only one copy running, but did not work correctly ...
- Spread in three ways
 - Exploited buffer overflow flaw in fingerd
 - Exploited flaw in sendmail debug mode
 - Guessing user passwords(!) on current network
- Bootstrap loader would be used to obtain the rest of the worm

21

Chernobyl virus (1998)

- When infected program run, virus becomes resident in memory of machine
 - Rebooting does not help
- Virus writes random garbage to hard drive
- Attempts to trash FLASH BIOS
 - Physically destroys the hardware...

22

Melissa virus/worm (1999)

- Word macro...
 - When file opened, would create and send infected document to names in user's Outlook Express mailbox
- Recipient would be asked whether to disable macros(!)
 - If macros enabled, virus would launch

23

Code red (2001)

- Propagated itself on web server running Microsoft's Internet Information Server
 - Infection using buffer overflow...
- Propagation by checking IP addresses on port 80 of the PC to see if they are vulnerable

24

Code Red I

- July 13, 2001: First worm of the modern era
- Exploited buffer overflow in Microsoft's Internet Information Server (IIS)
- 1st through 20th of each month: spread
 - Find new targets by random scan of IP address space
 - Spawn 99 threads to generate addresses and look for IIS
 - Creator forgot to seed the random number generator, and every copy scanned the same set of addresses ☹
- 21st through the end of each month: attack
 - Deface websites with "HELLO! Welcome to <http://www.worm.com>! Hacked by Chinese!"

25

Usurped Exception Handling In IIS

- Overflow in a rarely used URL decoding routine
 - A malformed URL is supplied to vulnerable routine...
 - ... another routine notices that stack has been smashed and raises an exception. Exception handler is invoked...
 - ... the pointer to exception handler is located on stack. It has been overwritten to point to a certain instruction inside the routine that noticed the overflow...
 - ... that instruction is CALL EBX. At that moment, EBX is pointing into the overwritten buffer...
 - ... the buffer contains the code that finds the worm's main body on the heap and executes it!

26

Code Red I v2

- July 19, 2001: Same codebase as Code Red I, but fixed the bug in random IP address generation
 - Compromised all vulnerable IIS servers on the Internet
 - Large vulnerable population meant fast worm spread
 - Scanned address space grew exponentially
 - 350,000 hosts infected in 14 hours!
- Payload: distributed packet flooding (denial of service) attack on www.whitehouse.gov
 - Coding bug causes it to die on the 20th of each month... but if victim's clock is wrong, resurrects on the 1st!
- Still alive in the wild!

27

Code Red II

- August 4, 2001: Same IIS vulnerability, completely different code, kills Code Red I
 - Known as "Code Red II" because of comment in code
 - Worked only on Windows 2000, crashed NT
- Scanning algorithm preferred nearby addresses
 - Chose addresses from same class A with probability $\frac{1}{2}$, same class B with probability $\frac{3}{8}$, and randomly from the entire Internet with probability $\frac{1}{8}$
- Payload: installed root backdoor in IIS servers for unrestricted remote access
- Died by design on October 1, 2001

28

Slammer (Sapphire) Worm

- January 24/25, 2003: UDP worm exploiting buffer overflow in Microsoft's SQL Server
 - Overflow was already known and patched by Microsoft... but not everybody installed the patch
- Entire code fits into a single 404-byte UDP packet
 - Worm binary followed by overflow pointer back to itself
- Classic buffer overflow combined with random scanning: once control is passed to worm code, it randomly generates IP addresses and attempts to send a copy of itself to port 1434
 - MS-SQL listens at port 1434

29

Slammer Propagation

- Scan rate of 55,000,000 addresses per second
 - Scan rate = rate at which worm generates IP addresses of potential targets
 - Up to 30,000 single-packet worm copies per second
- Initial infection was doubling in 8.5 seconds (!!)
 - Doubling time of Code Red was 37 minutes
- Worm-generated packets saturated carrying capacity of the Internet in 10 minutes
 - 75,000 SQL servers compromised
 - And that's in spite of broken pseudo-random number generator used for IP address generation

30

Slammer Impact

- \$1.25 Billion of damage
- Temporarily knocked out many elements of critical infrastructure
 - Bank of America ATM network
 - Entire cell phone network in South Korea
 - Five root DNS servers
 - Continental Airlines' ticket processing software
- The worm did not even have malicious payload... simply bandwidth exhaustion on the network and resource exhaustion on infected machines

31

Secret of Slammer's Speed

- Old-style worms (Code Red) spawn a new thread which tries to establish a TCP connection and, if successful, send a copy of itself over TCP
 - Limited by latency of the network
- Slammer was a connectionless UDP worm
 - No connection establishment, simply send 404-byte UDP packet to randomly generated IP addresses
 - Limited only by bandwidth of the network
- A TCP worm can scan even faster
 - Dump zillions of 40-byte TCP-SYN packets into link layer, send worm copy only if SYN-ACK comes back

32

Detecting viruses

- Can try to look for "signatures"
 - Unreliable unless up-to-date
 - Encrypted viruses
 - Polymorphic viruses
- Examine storage
 - Sizes of files, "jump" instruction at beginning of code
 - Can be hard to distinguish from normal software
- Check for (unusual) execution patterns
 - Hard to distinguish from normal software...

33

How hard is it to write a virus?

- 500 matches for "virus creation tool" in Spyware Encyclopedia

34