CS 392/681 - Computer Security

Nasir Memon
Module 9 – Malicious Logic
Course Policies and Logistics

- Reading list.
- Problem list.
- Homework 5 posted.
Malicious Logic

- So what does this code snippet do?

```c
#include <stdio.h>

int main(int argc, char** argv){
    while(1)
    {
        printf("\t\t\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b
Malicious Logic

- Malicious logic: "Hardware, software, or firmware capable of performing an unauthorized function on an information system." NSTISSI 4009
- Usually violates security policy of a system
- Malicious logic is also known as malicious code or malware
- Types of malicious logic?
Types of malicious logic

- **Virus** *(Vital Information Resources Under Siege)*
  - Self replicating code, parasitic

- **Trojan Horses**
  - Program with overt effects and covert effects

- **Worms**
  - Self replicating, spread through networks

- **Logic Bombs**
  - Waits for a trigger condition and “detonates”

- **Trapdoors**
  - Alternative means of executing code
    - Intentional – legitimate and malicious purposes
    - Exploits – exploits of faulty code, buffer overflow, format string
Types of malicious logic

- **ActiveX, Java code**
  - Execution of malicious code via Java applets, ActiveX scripts
  - Malicious mobile code

- **Bacteria and rabbits**
  - Program that absorbs all of some class of resource

- **Easter Eggs (http://www.eeggs.com/)**
  - Can we categorize them as malicious logic?
  - Does PowerPoint has Easter eggs? You’ll see 😊

- **Hybrids**
  - Usually a mixture of above
  - An Easter egg + Worm = Easter worm
Roadmap

- Virus
  - Types of viruses
  - Means of attaching
  - Anatomy of a simple virus
  - More sophisticated virus
  - Virus detection methods
  - Antivirus mechanisms
- Buffer overflow, format string vulnerabilities
- Further readings
Types of virus

- Classification by where they attach
  - Boot sector viruses
  - Parasitic viruses

- Classification by type of code
  - Binary viruses: usually written in assembly language then assembled to form executable image (binary file); attaches to other binary files or boot sector.
  - Macro viruses: written in high-level macro language then interpreted (possibly after pre-processing); attaches to other files that support same macro language (e.g. Outlook, Emacs)
Types of viruses

- A general classification
  - Boot sector viruses
    - Modify and reside in boot sector
  - Parasitic viruses
    - Attach itself to files
  - Polymorphic viruses
    - Mutate like biological viruses
  - Stealth Viruses
    - Hard to detect
  - TSRs (Terminate Stay Resident)
    - Memory resident viruses
  - LKMs (Loadable Kernel Modules)
    - Future of Unix based viruses
  - Multi-partite
    - Hybrids of above
Virus logic

- **Payload**
- **Trigger**
- **Propagation Engine**
- **Infection**
- **Mutation Engine**
- **Incubation Engine**

**Infection:** Infection is the act of replicating from a host to another host.

**Incubation:** The time between infection and activation of payload.

**Virulent:** Number of infections per copy.
Virus logic

- Virus includes code to
  - Search for files to infect
  - Replicate
    - Make copy of self
    - Attach to file/boot sector
  - Payload (check trigger and do badness)
  - Measures to allude detection
    - Ideally, should execute quickly then pass control to infected program’s normal code
    - Intercept system calls
    - Fool antiviral tools
Means of attaching: overwriting

- Virus overwrites an executable file
- Easiest mechanism
- Since original program is damaged easily detected
Means of attaching: at the beginning

- Improved stealth because original program is intact
- If original program is large, copying it may be slow
- File size grows if multiple infections occur
Means of attaching: intersperse

- Faster infection
- Harder to cleanup
Means of attaching: companions

- Sometimes exploit precedence of file extensions that is: .com → .exe → .bat
Memory residents or TSRs

- Infect memory-resident code (e.g. frequently used parts of the OS), which remains in memory while the computer is running
- Resident code usually activated many times, giving virus many opportunities to spread
- Example: attach to interrupt-handler and check whether any new floppies have been inserted; if so, infect boot sector
- De Tour: **Interrupts**
Mini-44: A simple COM infector

- Infects all COM files in current DOS directory by overwriting
- Background on execution of DOS COM files:

Typical DOS system call

- Set registers to indicate which ISR to call
- Set parameters to that ISR in appropriate registers
- Call interrupt 21
Anatomy of a simple virus

Uses DOS function 4E, Find First File Search, with argument that’s stored as ASCII string (including wild-card characters); writes identifying info to Disk Transfer Area (DTA) at location 80H in PSP (or returns an error code if no file found).

FNAME EQU 9EH
START:
    ;search for *.com
    mov    r1, 4E
    mov    r2, OFFSET COM_FILE
    int    21H
LP_SEARCH:...
DONE:   ret           ;return to DOS
COM_FILE:DB  ‘*.COM’,0
Anatomy of a simple virus

SEARCH_LP:

...  
  xchg r3, r4; put file handle in r4  
  mov r1, 40H  
  mov r5, 44; size of this virus  
  mov r2, 100H ; location of virus  
  int 21H  
...

Use DOS function 40H, Write to a File, with arguments file handle and size, location of virus code. Note that location of executable is known to be 100H in COM file.
Anatomy of a simple virus

SEARCH_LP:

... Mov r1,3EH ;close the file
int 21H

mov r1,4FH
int 21H ;search for next file
jmp SEARCH_LP

...

Uses DOS function 4FH, Find Next File Search, with argument that’s stored as ASCII string (including wildcard characters); writes identifying info to DTA or returns an error code if no file found.
Anatomy of a simple virus

SEARCH_LP:

jc DONE
mov r3,3D01H
mov r2,FNAME
int 21H

...

Use DOS function 3DH, Open File, with argument written to buffer by search, to open the file.
IVTs and ISRs

- Interrupt 21Hex is the gateway to system calls
- IVT maps interrupts to appropriate service routines
- IVT can be altered to point to user routines!
- ISRs look at registers for their parameters
Evaluation of Mini-44

- Simple enough to demonstrate concepts
- Not very effective:
  - overwriting: kills everything it infects, thus unlikely to spread without detection
  - only infects files in local directory
- Based on largely obsolete OS and file structure (DOS COM files)
- Still ... please don’t try to execute it.
Five major detection methods

- Integrity checking
  - Look for modified files by comparing old and new checksums
  - No software updates required
  - Requires maintenance of virus free checksums
  - Unable to detect passive, active stealth viruses
  - Cannot identify viruses by type or name

- Interrupt monitoring
  - Attempts to locate and prevent a viruses’ interrupt calls
  - Poor system utilization
  - Obstructive, because of false positives

- Memory detection
  - Depends on recognition of known viruses’ location and code in memory
Five major detection methods

- **Signature scanning**
  - Recognizes viruses’ unique “signature”: a pre-identified hex functional
  - Need to maintain current signature files and scanning engine refinements
  - False positives

- **Heuristics/Rule based**
  - Faster than traditional scanners
  - Uses a set of rules to effectively parse through files and identify code
  - Uses expert systems or neural networks
  - Depends on current rule-set

*(Detection can be performed on-access or on-demand)*
Properties of a good signature

- Should always appear in the virus, so there won’t be any false negatives
- Should not appear in (m)any other files, so there won’t be (m)any false positives
- Should be reasonably short, for efficient scanning
- For simple viruses like Mini-44, it’s easy to find good signatures. Virus writers have responded with...
Polymorphic Viruses

- Polymorphic = “many forms”
- Goal: Foil virus scanners by changing virus code each time virus replicates, so that it will be difficult to find a good signature
- Approaches:
  - Encrypt virus with random key
    - Note: Goals and techniques are different than in the encryption techniques we studied earlier. XOR with stored key is sufficient.
  - “Mutate” virus by making small changes that don’t affect the semantics of the code
  - Nearly 2 billion guises can be evolved from a single code
  - Requires algorithm based matching instead of simple string based matching
  - Given two code segments, evaluating their semantic equivalency is an undecidable problem!
Format of encrypted virus

Count = number of bytes in encrypted virus
L:   temp = fetch next byte
    temp = decrypt(temp)
    count--
    if count > 0 goto L
jump to start of decrypted code
data: value of decryption key
#$@($&)%^!*(_&%#)#$ ; encrypted
%(*$#&%)@(%$(&+^$@) ; virus code
...
Replication of encrypted virus

- Copy decryption engine to infected file (as is)
- Select new key and copy it to the infected file
- For each byte of the encrypted portion of the virus:
  - take decrypted byte
  - encrypt it with the new key
  - copy it to the infected file
- Result: different replicas of virus have different byte patterns, so difficult to find signature
Anti-virus tools’ answer to encryption

- Select the signature from the unencrypted portion of the code, I.e. the decryption engine

Problem:
- Anti-virus tools usually want to determine which virus is present, not just determine that some virus is present (in order to “disinfect”).
  - Can emulate the decryption then further analyze the decrypted code.
- Virus writers have responded by obscuring the encryption engine through mutations

- It’s a game of cat and mouse
Virus Analysis

- Analysis of virus by human expert
  - slow: by the time signature has been extracted, posted to AV tool database, downloaded to users, virus may have spread widely.
    - pre-1995: 6 months to a year for virus to spread world-wide
    - mid-90’s: a few months
    - now: days or hours
  - labor-intensive: too many new viruses
    - currently, 8-10 new viruses per day
  - can’t handle epidemics:
    - queue of viruses to be analyzed overflows
    - heavy demand on server that posts signatures & fixes

- Automated analysis, e.g. “Immune System”
  - developed at IBM Research
  - licensed to Symantec
Immune System Architecture

- Active network: controls "flooding"
- New virus
- Virus analysis center
- Signature and disinfection instructions
- Local administrators
Signature Extraction at VAC

- Virus allowed (encouraged) to replicate in controlled environment in immune center
- This yields collection of infected files
- In addition, a collection of “clean” files is available
- Machine learning techniques used to find strings that appear in most infected files and in few clean files, e.g.:
  - search files for candidate strings
    - add points if found in infected file
    - subtract points if found in clean file
Disinfection

- Once virus is detected, would like to clean up infected file
- AV tool must identify virus as well as detecting it
  - can then supply code to remove the virus
  - requires detailed understanding of how the particular virus attaches
Macro-viruses

- Written in macro-language, such as VBA
- Infect documents (as opposed to programs), such as word-processor docs, spreadsheets, etc.
- “Attach” by modifying commonly used macros
  - popular target is Normal.dot, which is opened when MS Office applications are executed
- Spread when documents are transmitted, via disks, file transfer, e-mail attachments, ...
- Macro virus dependencies:
  - Application popularity
  - Macro language depth
  - Macro implementation
  - Stubborn users
Melissa (W97M_Melissa)

- Fri. 3/26/99: initial reports of MS Word 97 and Word 2000 virus propagating via e-mail attachments
- Propagates when receiver opens attachment
- Initial document was list w/ references to pornographic web sites.
- Under certain circumstances, creates attachments with the victim’s docs [!!]
Melissa (W97M_Melissa)

- Lowers macro security settings to permit all macros to run when documents are opened in the future
- Checks/sets registry key to particular value so that virus will only propagate once per session
- Emailed copy of itself to first 50 entries in MS Outlook address book (some of which may be mailing lists)
- Infects Normal.dot template file
Effects of Melissa

- Denial of service on mail servers (due to exponential growth in number of e-mails being sent)
- If another document is opened after normal.dot has been infected, this document can be infected and will be attached to the e-mails that Melissa sends out
  - Leaks of sensitive information
Virus Prevention

- Install high-quality Anti-Virus software and update signatures frequently
- Beware of executable files from untrusted sources (including innocent victims of virus attacks)
- Beware of macros
Hoaxes

- Warnings of non-existent virus attacks with instructions to warn everyone you know
- Can result in flood of e-mails, possible denial of service
- Check with reliable sources, such as CERT, before [usually = instead of] forwarding such warnings
Means of exploiting a system

- **User to root:**
  - A local user on a system gets control of the system through privilege elevation.

- **Remote to user:**
  - An attacker on the network gains access to a user account on target host.

- **Remote to root:**
  - An attacker on the network gets control of the target host. No privilege elevation because attacker is not a user.

- **Remote disk read:**
  - An attacker on the network gains ability to read private data files on the target host without the authorization of the owner.

- **Remote disk write:**
  - An attacker on the network gains the ability to write to private data files on the target host without the authorization of the owner.
Buffer overflow

- Anecdotal notes suggest buffer overflows were known since sixties
- Proof of concept made public in 1988 by Morris Worm, which exploited finger daemon in Unix systems- Robert Morris
- CodeRed is the most recent and wide spread buffer overflow exploit, which exploited IIS web server (CodeRed infected nearly half a million hosts!)
- Take advantage of the lack of array bounds checking in C and C++ (and other languages) to transfer control to malicious code.
- Despite the fact that this vulnerability is well-known and preventable, buffer overflow attacks still prevalent
Buffer overflow

- Over 50% of the major security bugs leading to CERT Coordination Center advisories in 1999
Why buffer overflow is prevalent?

- **C/C++ is inherently unsafe?**
  - C/C++ library contains many “unsafe” functions
  - Pre, post conditions to these functions are not well understood

- **Lack of bound checking**
  - Programmers often forget to do bound checking
  - C/C++ don’t do bound checking, unlike Java
  - “Unsafe” functions lack bound checking

- **Code reuse**
  - Many unsafe libraries are heavily reused e.g. libc

- **Why are they so common on Unix flavors?**
  - Open source, therefore easy to exploit

- **Why they are not so common on Windows?**
  - Arguably, exploits for Windows are equality common even without source code
Vocabulary

- Program
  - An executable image on the disk

- Process
  - An executable image in the memory (ps –aux)

- Stack (Stack overflow?)
  - Operating system uses stacks to keep track of subroutine calls, store static variables and parameters

- Heap
  - Chunk of memory dynamically allocated at run time

- Buffer
  - A contiguous piece of memory

- Addressing (physical/logical)
  - How processes locate/identify instruction and data
Simple OS concepts

- Memory
- Devices
- Files
- Processes

Memory Tables
- I/O Tables
- File Tables

Process Tables
- Primary Process Table

Process Image
- User Data (rw)
- User Program (r)
- User Stack (r/w)
- PCB (r/w)

Process 1
- PID
- State Info
- Control Info
- Registers
- Visible-Registers
- Control info
- Stack pointer

Stack Base
- Stack Pointer
- Stack Limit

10/8.10/2002 Lecture 10 - Malicious Logic
Process stack

- Contiguous block of memory
- SP points to top of stack
- Bottom is fixed
- Kernel adjusts stack size dynamically
- Stack holds automatic variables, function parameters
- Grows downwards on most systems
- Stacks have logical segments called Stack Frames
  - Pushed when calling functions
  - Popped when returning from functions
  - Contains parameters, local variables, previous FP, IP
- Stack doesn’t overflow, buffers do
Procedure calls

```
Return
Old FP
x
y

P{
P
Q(m,n);
    x,y
    Q(m,n);
    }

Q(...){
    a,b
    }
```
Procedure calls cont...

```
push $0x2
push $0x3
call Q
push %ebp
movl %esp, %ebp
sub 0x8, %esp
```

Prolog:
- Push parameters
- Push return address
- Push FP
- Set current FP
- Reserve space for variables

Execution:

Epilog:
- Clear the stack
- Load old frame
- Setup IP with old IP
Exploiting buffer overflow

- Overflow a buffer allocated on the stack or on the heap in a way that causes the value of some important variable to change
  - e.g. flag indicating whether program can access private files
  - (harder w/ heap allocated arrays, since memory map can’t be predicted.)

- “Smash the stack”:
  - Overflow buffer allocated on the stack to change return pointer to the address of some malicious code
Simple exploit

```c
main(...){
    ...
    foo(m, str);
}

void foo(int x, char *str){
    char buf[8];
    for(i=0;i<x; ++i){
        buf = *str;
        ++buf; ++str;
    }
}
```

Exploit code: Also known as shell code, egg
Smashing the stack

- Find a stack-allocated buffer such that overflow will allow return address to be overwritten.
  - Requires detailed understanding of activation record layout
  - Use debugger or modify code to dump relevant addresses and their contents
- Place hostile code in memory.
- Write over return address to cause jump to hostile code.
Creating a memory map

void test (int i) {
    char b[12];
    printf("&i = %p\n", &i);
    printf("&b = %p\n", &b);
}

int main() {
    test(12);
}

• A good debugger saves lot of time
Look for return address

```c
char *j  //global to avoid changing stack
void test (int i)
{
    char b[12];
    printf("&main = %p\n",&main);
    for (j=b-8; j<((char*)&i)+8;j++)
        printf("%p: 0x%x\n",j,
                (unsigned char *)j);
}
int main()
{
    test(12);
}
```
Map stack frame

- Examine output to look for a byte sequence (4 bytes) representing address fairly close to &main.
- Need to be aware of details of byte order (big Endian vs. little Endian)
- create map of stack frame, e.g:
  - 0xbfffffa88-0xbfffffa93: char array b
  - 0xbfffffa94-0xbfffffa97: stack pointer
  - 0xbfffffa98-0xbfffffa9b: return address
  - 0xbfffffa9c-0xbfffffa9f: parameter i
Exploiting this knowledge

- Use memory map to decide how many bytes buffer should overflow by
- Run program with values that cause
  - malicious code to be stored at some known address, A
- Run program with input that copies value like

<table>
<thead>
<tr>
<th>exploit code</th>
<th>filler</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td>RET</td>
</tr>
</tbody>
</table>

- To the buffer
Inserting the Malicious Code

- Typical attack code on UNIX:

  ```c
  void exploit() {
      char *s = "/bin/sh";
      execl(s,s,0x00);
  }
  ```

- Compile to get executable code
- Figure out where string is stored
- Write the code somewhere: current stack frame, if it fits; elsewhere on stack; ...
Popular Targets

- Processes that run with higher privileges (e.g. process that runs with suid root)
  - Many UNIX functions need higher privileges to do things like writing to mail queue or opening a socket
  - Ordinary user programs may be temporarily granted higher privileges to invoke such functions
- Transfer control to code that spawns a shell
- Then use the shell (running with root privileges) to do whatever you want.
General effects of buffer overflow

- No effect (if overwritten variables are not subsequently accessed)
- Program crash
- Strange output (possibly difficult to reproduce)
- Malicious effects
  - if buffer overflow is carefully exploited to transfer control to malicious code
- Note that buffer overflows may be difficult to detect with conventional software testing.
Advanced topics

- Heap overflows
  - Overflows dynamic memory
  - Much harder than smashing the stack but works
- Virtual pointer smacks
  - Works on object oriented programs
  - Manipulates virtual pointer instead of return address
- Code alignment with NOP
  - Exploit code must be aligned properly in the buffer with return address pointing to right place
  - Exploit code is usually padded with NOPs at the beginning and return address points to this NOPs to make the probability of hitting exploit code higher
- Polymorphic code
  - To evade IDS in remote exploits
  - Just like polymorphic viruses
- Backbox exploits
  - Exploiting programs for which no source code is available
Windows vs (open source) UNIX

- **Windows:**
  - Buffer overflows exploited to install trap-door, such as Back Orifice
  - "Security by obscurity"
  - Harder because source code is not available but possible

- **Unix:**
  - Buffer overflows usually exploited to create interactive shell
  - Security by "many eyes"
  - OpenBSD is audited manually for buffer overflows

- Experts generally believe that Windows has more exploitable buffer overflow problems than UNIX
Buffer overflow defenses

- Do bound checking where necessary
  - The overhead is small compared to the risks
  - Write daemons in a type-safe language like Java
- Using dynamic memory (heap) for buffers does not solve the problem. Only makes it harder to exploit
- Integrate security into software developments
  - Code reviews
  - Testing
  - Checking security of added components
Buffer overflow defenses

- Some unsafe functions in C library
  - strcpy()
  - strcmp()
  - strcat()
  - strlen()
  - gets()
  - fgets()
  - ...

- Avoid these functions in your programs and use companions such as strncpy(), strncmp(), strncat() etc.
Buffer overflow defenses

- Non-executable buffers
  - Make data segment of address space non-executable. Although this is how older operating systems were designed, newer versions of Unix and Windows make the data segment executable for performance optimization.
  - Signal delivery
  - Trampoline functions
  - LISP functions

- Bound checking by compilers
  - Compilers can insert bound checking code at compile time e.g. Compaq C Compiler
  - Library functions are not checked
  - Code might be too complicated to automate
Buffer overflow defenses

- Type-safe Languages
  - If type-safe operations can be performed on a given variables then arbitrary changes cannot be used to execute code

- Stack integrity checks
  - StackGuard: Compiler generates integrity checks on activation records
  - PointGuard: Compiler generates integrity checks on instruction pointer
Guarding against malicious code

- Good software engineering practice
  - code reviews
  - careful independent testing
  - proof of correctness
- Operating system controls
  - restricted access
  - audit logs
  - etc
Further Reading

- [www.cert.org](http://www.cert.org)