Topics in Security Testing

Threat modeling

- Threat Modeling is a process for evaluating a software system for security issues.
- It is a variation of the code and specification inspections processes discussed earlier in the course.
- The goal is for a review team to look for software features that vulnerable from a security perspective.
- Threat modeling is not the responsibility of a software tester, although testers may be involved in the security review team.

Threat modeling process (1)

- Assemble the treat modeling team
  - Include security experts and consultants
- Identify the assets
  - E.g., credit card numbers, social security numbers, computing resources, trade secrets, financial data
- Create an architecture overview
  - Define the architecture and identify the trust boundaries and authentication mechanisms
- Decompose the application
  - E.g., identify data flows, encryption processes, password flows.

Threat modeling process (2)

- Identify threats
  - E.g., can data be viewed, changed? Limit access of legitimate users? Unauthorized access of the system?
- Document threats
  - E.g., describe threat, target, form of attack, counter-measures to prevent an attack, etc.
- Rank threats (scale: low, medium, high)
  - Damage potential
    - E.g., property, data integrity, financial loss
  - Reproducibility
    - E.g., probability that an attempt to compromise the system will succeed
  - Exploitability/Discoverability
    - E.g., is it difficult to hack into the system?
  - Affected users
    - How many users will be affected? Who are these users? Are they important?

What is Malware?

- Malware (malicious software) is any program that works against the interest of the system's user or owner.
- **Question:** Is a program that spies on the web browsing habits of the employees of a company considered malware?
- What if the CEO authorized the installation of the spying program?

Uses of Malware

- Why do people develop and deploy malware?
  - Financial gain
  - Psychological urges and childish desires to “beat the system”.
  - Access private data
  - …

Typical purposes of Malware

- Backdoor access:
  - Attacker gains unlimited access to the machine.
- Denial-of-service (DoS) attacks:
  - Infect a huge number of machines to try simultaneously to connect to a target server in hope of overwhelming it and making it crash.
- Vandalism:
  - E.g., defacing a web site.
- Resource Theft:
  - E.g., stealing other user's computing and network resources, such as using your neighbors' Wireless Network.
- Information Theft:
  - E.g., stealing other user's credit card numbers.

Types of Malware

- Viruses
- Worms
- Trojan Horses
- Backdoors
- Mobile code
- Adware
- Sticky software

Viruses

- Viruses are self-replicating programs that usually have a malicious intent.
- Old fashioned type of malware that has become less popular since the widespread use of the Internet.
- The unique aspect of computer viruses is their ability to self-replicate.
- However, someone (e.g., user) must execute them in order for them to propagate.

Viruses (Cont'd)

- Some viruses are harmful (e.g.,):
  - delete valuable information from a computer's disk,
  - freeze the computer.
- Other viruses are harmless (e.g.,):
  - display annoying messages to attract user attention,
  - just replicate themselves.

Viruses: Operation

- Viruses typically attach themselves to executable program files
  - e.g., .exe files in MS Windows
- Then the virus slowly duplicates itself into many executable files on the infected system.
- Viruses require human intervention to replicate.

Origin of the term computer virus

- The term computer *virus* was first used in an **academic** publication by Fred Cohen in his 1984 paper *Experiments with Computer Viruses*.
- However, a mid-1970s science fiction novel by David Gerrold, *When H.A.R.L.I.E. was One*, includes a description of a fictional computer program called VIRUS.
- John Brunner's 1975 novel The *Shockwave Rider* describes programs known as *tapeworms* which spread through a network for deleting data.
- The term computer virus also appears in the comic book *Uncanny X-Men* in 1982.

The first computer viruses

- A program called *Elk Cloner* is credited with being the first computer virus to appear "in the wild". Written in 1982 by Rich Skrenta, it attached itself to the Apple DOS 3.3 operating system and spread by floppy disk.
- The first PC virus was a boot sector virus called (c)Brain, created in 1986 by two brothers, Basit and Amjad Farooq Alvi, operating out of Lahore, Pakistan.

Worms

- Worms are malicious programs that use the Internet to spread.
- Similar to a virus, a worm self-replicates.
- Unlike a virus, a worm does not need human intervention to replicate.
- Worms have the ability to spread uncontrollably in a very brief period of time.
  - Almost every computer system in the world is attached to the same network.

Worms: Operation

- A worm may spread because of a software vulnerability exploit:
  - Takes advantage of the OS or an application program with program vulnerabilities that allow it to hide in a seemingly innocent data packet.
- A worm may also spread via e-mail.
  - Mass mailing worms scan the user's contact list and mail themselves to every contact on such a list.
  - In most cases the user must open an attachment to trigger the spreading of the worm (more like a virus).

Trojan horses

- A Trojan Horse is a seemingly innocent application that contains malicious code that is hidden somewhere inside it.
- Trojans are often useful programs that have unnoticeable, yet harmful, side effects.

Trojan horses: Operation (1)

- Embed a malicious element inside an otherwise benign program.
- The victim:
  1. receives the infected program,
  2. launches it,
  3. remains oblivious of the fact that the system has been infected.
- The application continues to operate normally to eliminate any suspicion.

Trojan horses: Operation (2)

- Fool users into believing that a file containing a malicious program is really an innocent file such as a video clip or an image.
- This is easy to do on MS Windows because file types are determined by their extension as opposed to examining the file headers.
- E.g.,
  - “A Great Picture.jpg .exe”
  - The .exe might not be visible in the browser.
  - The Trojan author can create a picture icon that is the default icon of MS Windows for .jpg files.

Backdoors

- A backdoor is malware that creates a covert access channel that the attacker can use for:
  - connecting,
  - controlling,
  - spying,
  - or otherwise interacting with the victim's system.

Backdoors: Operation

- Backdoors can be embedded in actual programs that, when executed, enable the attacker to connect to and to use the system remotely.
- Backdoors may be planted into the source code by rogue software developers before the product is released.
  - This is more difficult to get away with if the program is open source.

Mobile code

- Mobile code is a class of benign programs that are:
  - meant to be mobile,
  - meant to be executed on a large number of systems,
  - not meant to be installed explicitly by end users.
- Most mobile code is designed to create a more active web browsing experience.
  - E.g., Java applets, ActiveX controls.

Mobile code (Cont'd)

- Java scripts are distributed in source code form making them easy to analyze.
- ActiveX components are conventional executables that contain native IA-32 machine code.
- Java applets are in bytecode form, which makes them easy to decompile.

Mobile code: Operation

- Web sites quickly download and launch a program on the end user's system.
- User might see a message that warns about a program that is about to be installed and launched.
  - Most users click OK to allow the program to run.
  - They may not consider the possibility that malicious code is about to be downloaded and executed on their system.

Adware

- Adware is a program that forces unsolicited advertising on end users.
- Adware is a new category of malicious programs that has become very popular.
- Adware is usually bundled with free software that is funded by the advertisements displayed by the Adware program.

Adware: Operation (1)

- The program gathers statistics about the end user's browsing and shopping habits.
  - The data might be transferred to a remote server.
- Then the Adware uses the information to display targeted advertisements to the end user.

Adware: Operation (2)

- Adware can be buggy and can limit the performance of the infected machine.
  - E.g., MS IE can freeze for a long time because an Adware DLL is poorly implemented and does not use multithreading properly.
- Ironically, buggy Adware defeats the purpose of the Adware itself.

Sticky software

- Sticky software implements methods that prevent or deter users from uninstalling it manually.
- One simple solution is not to offer an uninstall program.
- Another solution in Windows involves:
  - installing registry keys that instruct Windows to always launch the malware as soon as the system is booted.
  - The malware monitors changes to the registry and replace the keys of they are deleted by the user.
  - The malware uses two mutually monitoring processes to ensure that the user does not terminate the malware before deleting the keys.

Future Malware

- Today's malware is just the tip of the iceberg.
- The next generation of malware may take control of the low levels of the computer system (e.g., BIOS, Firmware).
  - The antidote software will be in the control of the malware …
- Also the theft of valuable information can result in holding it for ransom.

Information-stealing worms

- Present-day malware does not take advantage of cryptography much.
- Asymmetric encryption creates new possibilities for the creation of information-stealing worms.
- A worm encrypts valuable data on the infected system using an asymmetric cipher and hold the data as ransom.

Information-stealing worms:Operation

1. The Kleptographic worm embeds a public encryption key in its body.
2. It starts encrypting every bit of valuable data on the host using the public key.
3. Decryption of the data is impossible without the private key.
4. Attacker blackmails the victim demanding ransom.
5. Attacker exchanges the private key for the ransom while maintaining anonymity.
   - Theoretically possible using zero-knowledge proofs
   - Attacker proves that he has the private key without exposing it.

BIOS/Firmware Malware

- Antivirus programs assume that there is always some trusted layer of the system.
- Naïve antivirus programs scan the hard drive for infected files using the high-level file-system service.
- A clever virus can intercept file system calls and present to the virus with fake versions (original/uninfected) of the files on disk.
- Sophisticated antivirus programs reside at a low enough level (in OS kernel) so that malware cannot distort their view of the system.

BIOS/Firmware Malware: Operations (1)

- What is the malware altered an extremely low level layer of the system?
- Most CPUs/hardware devices run very low-level code that implements each assembly language instruction using low level instructions (micro-ops).
- The micro-ops code that runs inside the processor is called firmware.
- Firmware can be updated using a firmware-updating program.

BIOS/Firmware Malware: Operations (2)

- Malicious firmware can (in theory) be included in malware that defeats antivirus programs.
- The hardware will be compromised by the malicious firmware.
- Not easy to do in practice because firmware update files are encrypted (private key inside the processor).

Antivirus programs

- Antivirus programs identify malware by looking for unique signatures in the code of each program (i.e., potential virus) on a computer.
  - A signature is a unique sequence of code found in a part of the malicious program.
- The antivirus program maintains a frequently updated database of virus signatures.
  - The goal is for the database to contain a signature for every known malware program.
- Well known antivirus software includes:
  - Symantec (http://www.symantec.com)
  - McAfee (http://www.mcafee.com)

Polymorphic viruses

- Polymorphism is a technique that thwarts signature-based identification programs.
- Polymorphic viruses randomly encode or encrypt the program code in a semantics-preserving way.
- The idea is to encrypt the code with a random key and decrypt it at runtime.
  - Each copy of the code is different because of the use of a random key.

Polymorphic viruses: Decryption technique

- A decryption technique that polymorphic viruses employ involves “XORing” each byte with a randomized key that was saved by the parent virus.
- The use of XOR-operations has the additional advantage that the encryption and decryption routine are the same:
  - a xor b = c
  - c xor b = a

Polymorphic viruses: Weaknesses

- Many antivirus programs scan for virus signatures in memory.
  - I.e., after the polymorphic virus has been decrypted.
- If the virus code that does the decryption is static, then the decryption code can be used as a signature.
- This limitation can be addressed (somewhat) if the decryption code is scrambled (superficially):
  - randomize the use of registers,
  - add no-ops in the code, …

Metamorphic viruses

- Instead of encrypting the program's body and making slight alterations in the decryption engine, alter the entire program each time it is replicated.
- This makes it extremely difficult for antivirus writers to use signature-matching techniques to identify malware.
- Metamorphism requires a powerful code analysis engine that needs to be embedded into the malware.

Metamorphic viruses: Operation

- Metamorphic engine scans the code and generates a different version of it every time the program is duplicated.
- The metamorphic engine performs a wide variety of transformations on the malware and on the engine itself.
  - Instruction and register randomization.
  - Instruction ordering
  - Reversing (negating) conditions
  - Insertion of “garbage” instructions
  - Reordering of the storage location of functions

Timeline of famous malware (1982-1988) [wikipedia]

- 1982
  - **Elk Cloner**, written for Apple II systems, is credited with being the first computer virus.
- 1987
  - **(c)Brain**, the first virus written for PCs.
  - **SCA**, a boot sector virus for Amiga appears, immediately creating a pandemic virus-writer storm. A short time later, SCA releases another, considerably more destructive virus, the Byte Bandit.
- 1988
  - **Morris** worm infects DEC VAX machines connected to the Internet, and becomes the first worm to spread extensively.

Timeline of famous malware (1998-2000) [wikipedia]

- 1998
  - **CIH** virus version 1.
- 1999
  - **Melissa** worm is released, targeting Microsoft Word and Outlook-based systems, and creating considerable network traffic.
- 2000
  - The **VBS/Loveletter** worm, also known as the "I love you" virus appeared. As of 2004, this was the most costly virus to business, causing upwards of 10 billion dollars in damage.

Timeline of famous malware (2001) [wikipedia]

- **Klez** worm.
- **Nimda** worm.
- **Code Red II** worm (spreads in China, attacks Microsoft's Internet Information Services.
- **Sircam** worm (spreads through e-mails and unprotected network shares).
- **Sadmind** worm (spreads by exploiting holes in both Sun Microsystem's Solaris and MS IIS).
- **Raman** worm (similar to the Morris worm infected only Red Hat Linux machines running version 6.2 and 7.0, using three vulnerabilities in `wu-ftpd`,`rpc-statd` and `lpd`.

Timeline of famous malware (2003) [wikipedia]

- **Sober** worm is first seen and maintains its presence until 2005 with many new variants.
- **Sobig** worm (technically the Sobig.F worm) spread rapidly via mail and network shares.
- **Blaster** worm also know as the Lovesan worm spread rapidly by exploiting MS computers.
- **SQL slammer** worm also known as the Sapphire worm, attacked vulnerabilities in Microsoft SQL Server and MSDE, causes widespread problems on the Internet.

Timeline of famous malware (2004) [wikipedia]

- **Sasser** worm emerges by exploiting a vulnerability in LSASS, causes problems in networks.
- **Witty** worm is a record breaking worm in many regards.
  - It exploited holes in several Internet Security Systems (ISS) products.
  - it was the first internet worm to carry a destructive payload and it spread rapidly using a pre-populated list of ground-zero hosts.
- **MyDoom** emerges, and currently holds the record for the fastest-spreading mass mailer worm.

Timeline of famous malware (2005) [wikipedia]

- **Zotob** worm, the effect was overblown because several United States media outlets were infected.

Software vulnerability

- What are software vulnerabilities?
- Types of vulnerabilities
  - E.g., Buffer Overflows
- How to find these vulnerabilities and prevent them?
- Classes of software vulnerabilities
- A software vulnerability is an instance of a fault in the specification, development, or configuration of software such that its execution can violate the (implicit or explicit) security policy.

Types of software vulnerability

- Buffer overflows
  - Smash the stack
  - Overflows in `setuid` regions
- Heap overflows
- Format string vulnerabilities

What is a buffer?

- Example:
  - A place on a form to fill in last name where each character has one box.
- “Buffer” is used loosely to refer to any area of memory where more than on piece of data is stored.

Buffer overflows

- The most common form of security vulnerability in the last 10 years
  - 1998: 2 out of 5 “remote to local” attacks in Lincoln Labs Intrusion Detection Evaluation were buffer overflows.
  - 1998: 9 out of 13 CERT advisories involved buffer overflows.
  - 1999: at least 50% of CERT advisories involved buffer overflows.

How does a buffer overflow happen?

- Reading or writing past the end of the buffer → overflow
- As a result, any data that is allocated near the buffer can be read and potentially modified (overwritten)
  - A password flag can be modified to log in as someone else.
  - A return address can be overwritten so that it jumps to arbitrary code that the attacker injected (smash the stack) → attacker can control the host.

Two steps

- Arrange for suitable code to be available in the program's address space (buffer)
  - Inject the code
  - Use code that is already in the program
- Overflow the buffer so that the program jumps to that code.

Inject the code

- Use a string as input to the program which is then stored in a buffer.
- String contains bytes that are native CPU instructions for attacked platform.
- Buffer can be located on the stack, heap, or in static data area.

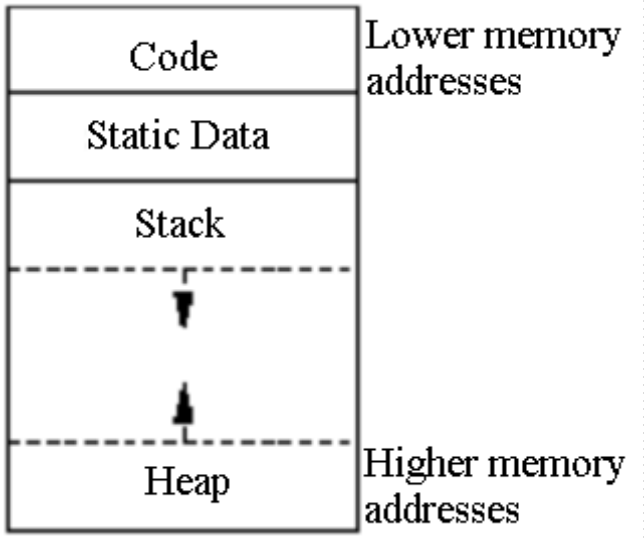
Code already in program

- Only need to parameterize the code and cause the program to jump to it.
- Example:
  - Code in libc that executes "exec(arg)", where arg is a string pointer argument, can be used to point to "/bin/sh" and jump to appropriate instructions in libc library.

Jump to attack code

- Activation record
  - stack smashing attack
- Function pointer
- Longjpm(3) buffer

Memory regions



Code/text segment

- Static
- Contains code (instructions) and read-only data
- Corresponds to `text` section of executable file
- If attempt to write to this region → segmentation violation

Data segment

- Permanent data with statically known size
- Both initiated and uninitiated variables
- Corresponds to the data-bss sections of the executable file
- brk(2) system call can change data segment size
- Not enough available memory → process is blocked and rescheduled with larger memory

Heap

- Dynamic memory allocation
- malloc() in C and new in C++ → More flexibility
- More stable data storage – memory allocated in the heap remains in existence for the duration of a program
- Data with unknown lifetime – global (storage class external) and static variables

Stack – I

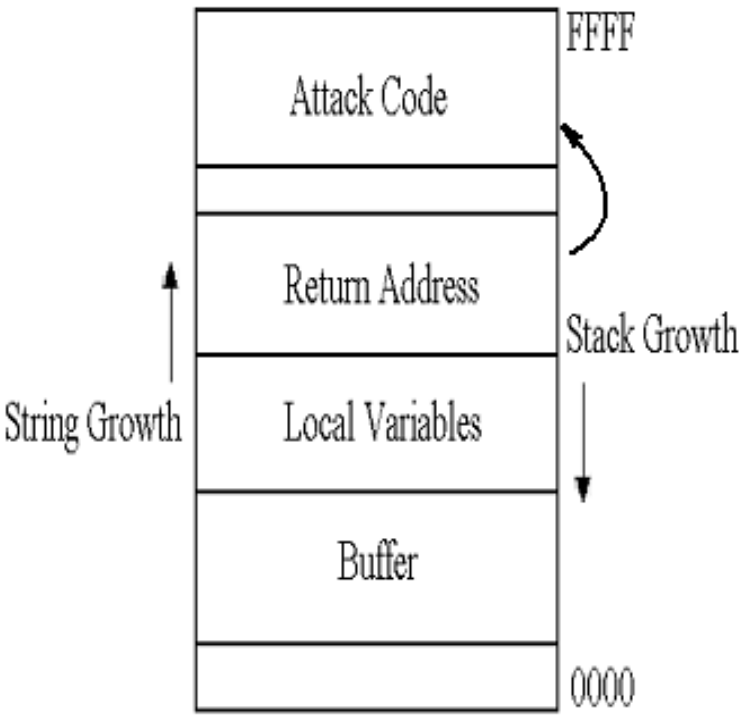
- Provides high-level abstraction
  - Allocates local variables when a function gets called (with known lifetime)
  - Passes parameters to functions
  - Returns values from functions
- Push/Pop operations (LIFO) – implemented by CPU
- Size – dynamically adjusted by kernel at runtime

Stack – II

- Stack Pointer (SP) – TOP of stack (or next free available address)
- Fixed address – BOTTOM of stack
- Logical Stack Frame (SF) – contains parameters to functions, local variables, data to recover previous SF (e.g: instruction pointer at time of function call)
- Frame Pointer (FP)/local Base Pointer (BP) – Beginning of Activation Record (AR), used for referencing local variables and parameters (accessed as offsets from BP)

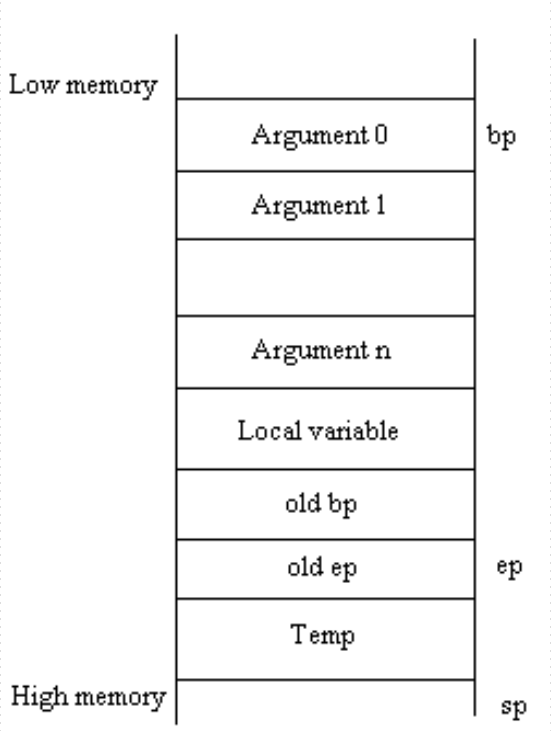
Activation record

- Contains all info local to a single invocation of a procedure
  - Return address
  - Arguments
  - Return value
  - Local variables
  - Temp data
  - Other control info



Accessing an activation record

- Base pointer: beginning of AR
  - Arguments are accessed as offsets from bp
- Environment pointer: pointer to the most recent AR (usually a fixed offset from bp)
- Stack pointer: top of AR stack
  - Temporaries are allocated on top on stack



When a procedure is called

- Previous FP is saved
- SP is copied into FP → new FP
- SP advances to reserve space for local variables
- Upon procedure exit, the stack is cleaned up

Function pointer

- Find a buffer adjacent to function pointer in stack, heap or static data area
- Overflow buffer to change the function pointer so it jumps to desired location
- Example: attack against superprobe program - Linux

Longjpm buffer

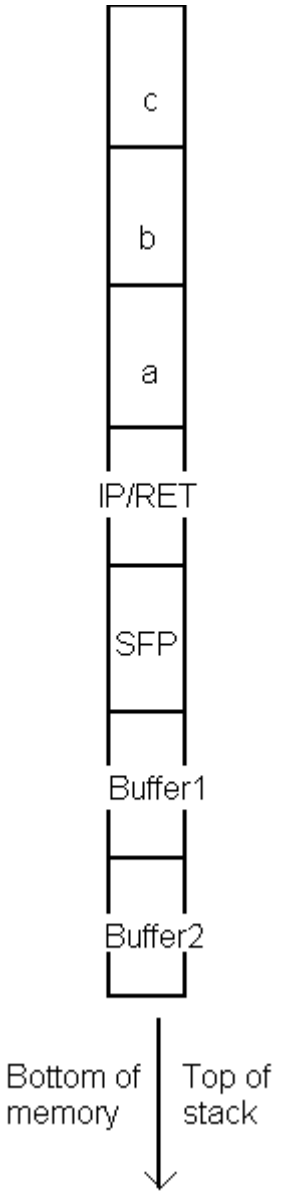
- setjmp(buffer) to set a checkpoint
- longjmp(buffer) to go back to checkpoint
- Corrupt state of buffer so that longjmp(buffer) jumps to the attack code instead

Example

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}
void main() {
  function(1,2,3);
}
```

```
pushl $3
pushl $2
pushl $1
call function

pushl %ebp
movl %esp,%ebp
subl $20,%esp
```



Buffer overflow example

```
void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
void function(int a, int b,
    int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = buffer1 + 12;
  (*ret) += 8; }
```

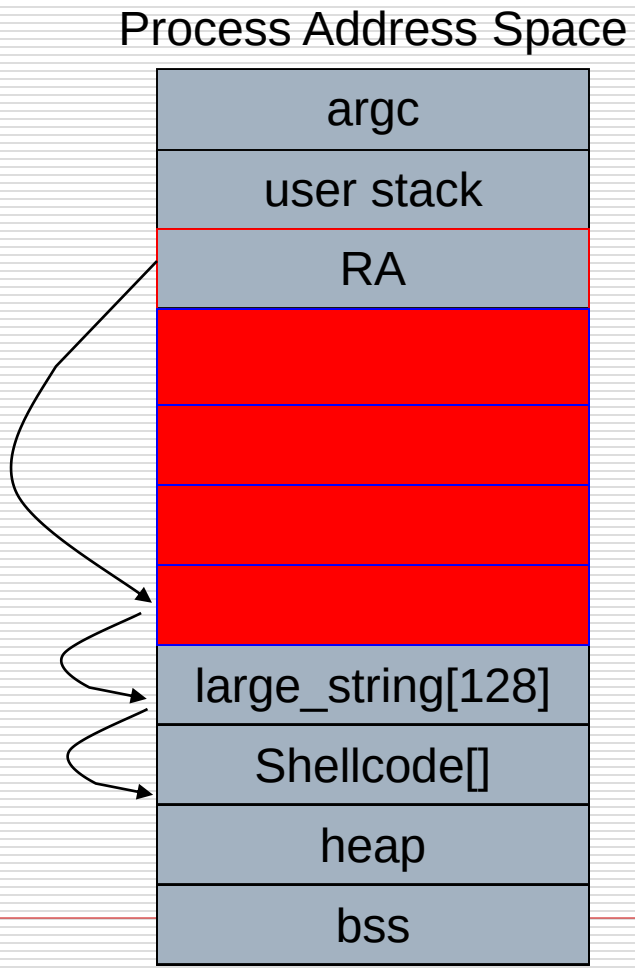
Result of program

- Output: 0
- Return address has been modified and the flow of execution has been changed
- All we need to do is place the code that we are trying to execute in the buffer we are overflowing, and modify the return address so it points back to buffer

Example [6]

```
char shellcode[ ] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
      char buffer[96];
      int i;
      long *long_ptr = (long *) large_string; /* long_ptr takes the address of large_string */
      /* large_string's first 32 bytes are filled with the address of buffer */
      for (i = 0; i < 32; i++)
            *(long_ptr + i) = (int) buffer;
      /* copy the contents of shellcode into large_string */
      for (i = 0; i < strlen(shellcode); i++)
            large_string[ i ] = shellcode[ i ];
      /* buffer gets the shellcode and 32 pointers back to itself */
      strcpy(buffer, large_string); }
```

Example illustrated [6]



Buffer overflows defenses

- Writing correct code (good programming practices)
- Debugging Tools
- Non-executable buffers
- Array bounds checking
- Code pointer integrity checking (e.g., StackGuard)

Problems with C

- Some C functions are problematic
  - Static size buffers
  - Do not have built-in bounds checking
- While loops
  - Read one character at a time from user input until end of line or end of file
  - No explicit checks for overflows

Some problematic C functions

Function	Severity	Solution: Use
gets	Most Risky	fgets(buf, size, stdin)
strcpy, strcat	Very Risky	strncpy, strncat
sprintf, vsprintf	Very Risky	snprintf, vsnprintf or precision specifiers
scanf family	Very Risky	precision specifiers or do own parsing
realpath, syslog	Very Risky (depending on implementation)	Maxpathlen and manual checks
getopt, getopt_long, getpass	Very Risky (depending on implementation)	Truncate string inputs to reasonable size

Good programming practices – I (useful to know for code inspections)

DO NOT USE:	Instead USE:

void main( ) {
char buf [40];
gets(buf);
}
	void main( ) {
	char buf [40];
	fgets(buf,40,stdin);
	}

Good programming practices – II

DO NOT USE:	Instead USE:

void main() {
char buf[4];
char src[8] = "rrrrr";
strcpy(buf,src);
}
	if (src_size >= buf_size) {
	cout<< "error";
	return(1);
	}
	else {
	strcpy(buf,src);
	}
	OR
	strncpy(buf,src,buf_size - 1);
	buf[buf_size - 1] = '\0';

Debugging tools

- More advanced debugging tools
  - Fault injection tools – inject deliberate buffer overflow faults at random to search for vulnerabilities
  - Static analysis tools – detect overflows
- Can only minimize the number of overflow vulnerabilities but cannot provide total assurance

Non-executable buffers

- Make data segment of program's address space non-executable → attacker can't execute code injected into input buffer (compromise between security and compatibility)

Non-executable buffers

- If code already in program, attacks can bypass this defense method
- Kernel patches (Linux and Solaris) – make stack segment non-executable and preserve most program compatibility

Array bounds checking

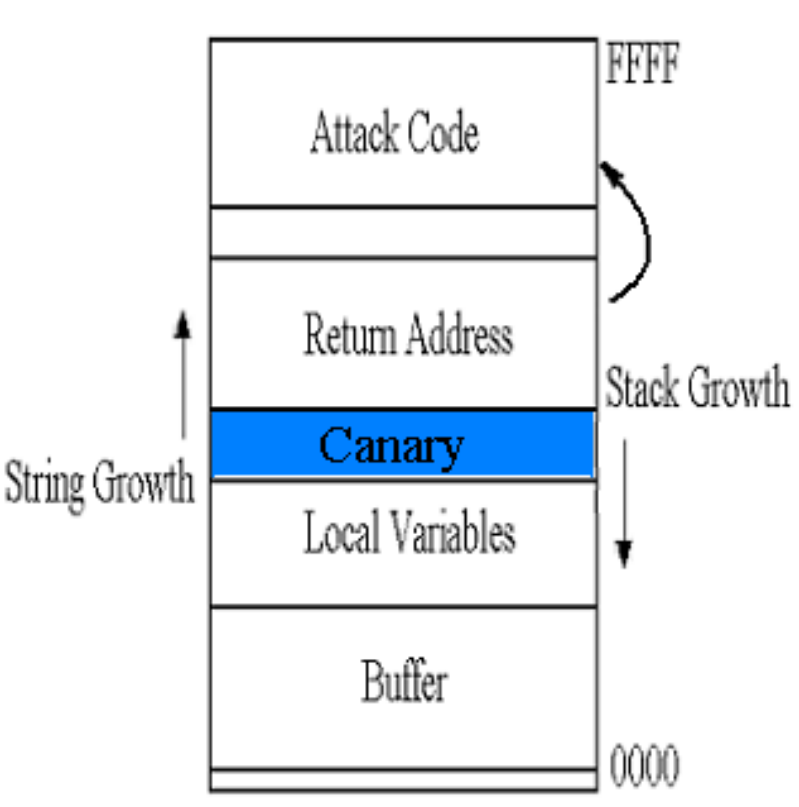
- Attempts to prevent overflow of code pointers
- All reads and writes to arrays need to be checked to make sure they are within bounds (check most array references)
  - Campaq C compiler
  - Jones & Kelly array bound checking
  - Purify memory access checking
  - Type-safe languages (e.g., Java)

Code pointer integrity checking

- Attempts to detect that a code pointer has been corrupted before it is de-referenced
- Overflows that affect program state components other than code pointer will succeed
- Offers advantages in performance, compatibility with existing code and implementation effort
  - Hand-coded stack introspection
  - StackGuard → PointGuard

StackGuard

- Compiler technique that provides protection by checking the return address in AR
- When detects an attack → causes app to exit, rather than yielding control to attacker
  - Terminator canary
  - Random canary



Heap overflows

- Harder to exploit, yet still common
- Need to know which variables are security critical
- Cause a buffer overflow to overwrite the target variables (generally buffer needs to have lower address)

Example

```
void main(int argc, char **argv) {
        char *super_user = (char *)malloc(sizeof(char)*9);
        char *str = (char *)malloc(sizeof(char)*4);
        char *tmp;
        super_user = super_user - 40;
        strcpy(super_user, "viega");
        if (argc > 1)
                strcpy(str, argv[1]);
        else
                strcpy(str,"xyz");
        tmp = str;
        while(tmp <= super_user + 12) {
        printf("%p: %c (0x%x)\n", tmp, isprint(*tmp) ? *tmp : '?', (unsigned int)(*tmp));
        tmp+=1; } }
```

Output

```
Address of str is: 003000E0
Address of super_user is: 00300110
003000E0: x (0x78)
003000E1: y (0x79)
003000E2: z (0x7a)
003000E3: ? (0x0)
003000E4: ² (0xfffffffd)
003000E5: ² (0xfffffffd)
003000E6: ² (0xfffffffd)
003000E7: ² (0xfffffffd)
003000E8: v (0x76)
003000E9: i (0x69)
003000EA: e (0x65)
003000EB: g (0x67)
003000EC: a (0x61)
003000ED: ? (0x0)
003000EE: ? (0x0)
003000EF: ? (0x0)
003000F0: h (0x68)
003000F1: I (0x49)
003000F2: / (0x2f)
003000F3: ? (0x0)
003000F4: ? (0xffffffc0)
```

Output after overflow

```
C:\Documents and Settings\Maralle\Buffer_Overflow\Example2\Debug>example2 xyz...
..maral
Address of str is: 003000D0
Address of super_user is: 00300100
003000D0: x (0x78)
003000D1: y (0x79)
003000D2: z (0x7a)
003000D3: . (0x2e)
003000D4: . (0x2e)
003000D5: . (0x2e)
003000D6: . (0x2e)
003000D7: . (0x2e)
003000D8: m (0x6d)
003000D9: a (0x61)
003000DA: r (0x72)
003000DB: a (0x61)
003000DC: l (0x6c)
003000DD: ? (0x0)
003000DE: ? (0x0)
003000DF: ? (0x0)
003000E0: h (0x68)
003000E1: I (0x49)
003000E2: / (0x2f)
003000E3: ? (0x0)
003000E4: ? (0xffffffb0)
```

Auditing for software security

- Pre-release software development practices unlikely to change
  - Safe languages
  - Adequate software design
  - Thorough testing
- Post-release auditing typically used if warranted

Security auditing problems

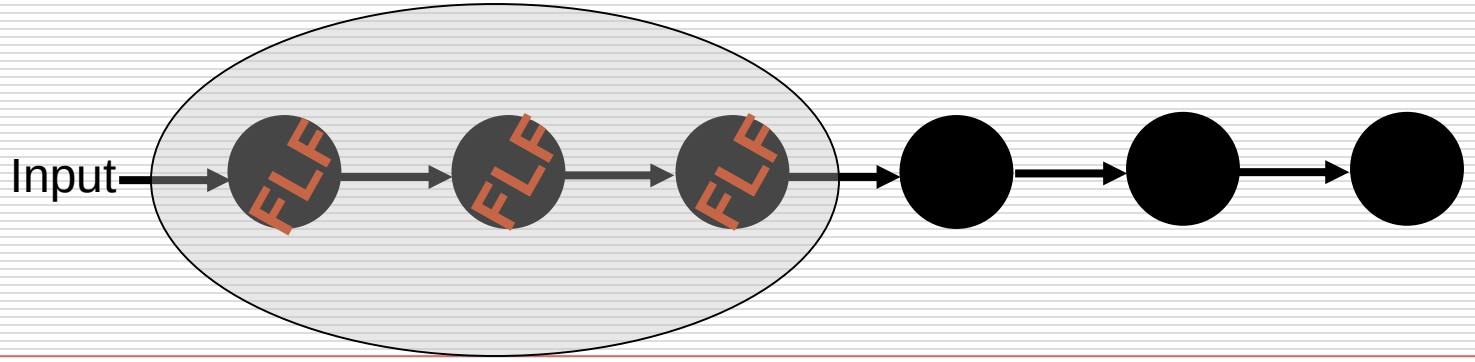
- Large scale auditing infeasible due code size
- Good source code will have 1-3 bugs for every 100 lines of code
- Security auditors need to find the software security vulnerabilities in the bugs
- Security audits would benefit from a tool that identify areas that are likely vulnerable

Improving the security audit

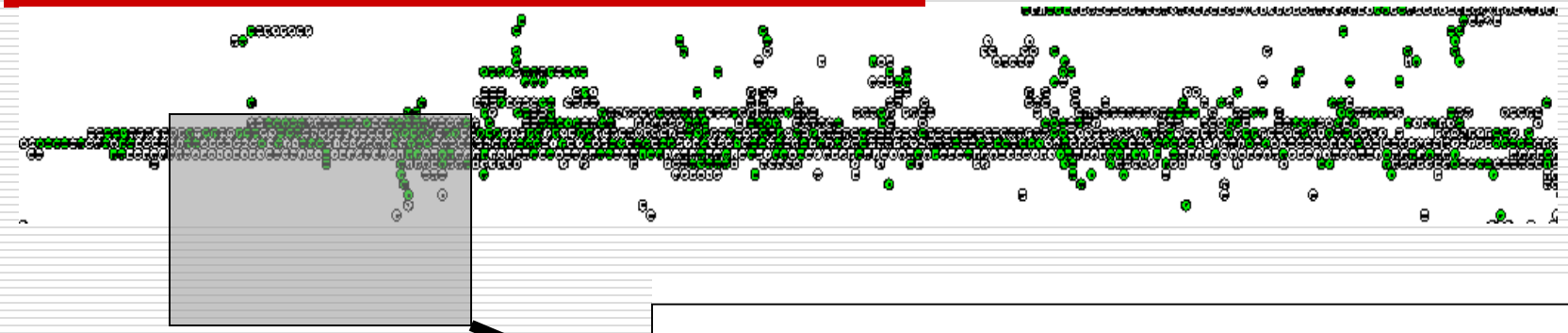
- Cut down the amount of code needed to be reviewed
- Ensure high degree of accuracy
- Bottom line: Complexity reduction

The FLF hypothesis

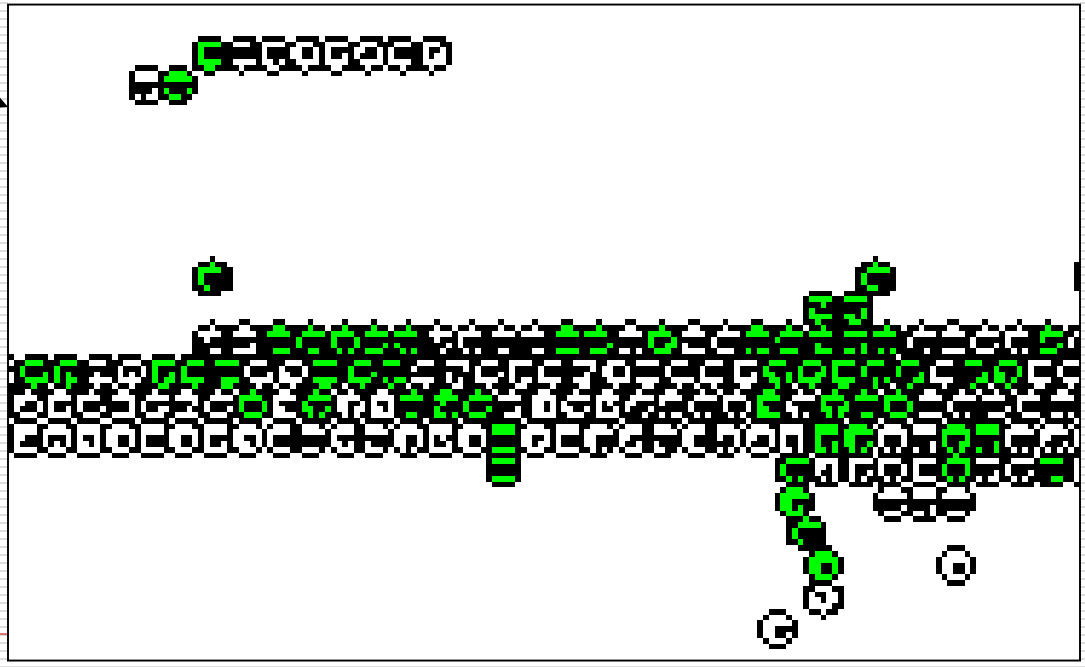
- A small percentage of functions near a source of input are more likely to contain software security vulnerabilities
- These functions are known as Front Line Functions



Front line functions



- 60% Code Reduction
- No auditor time required
- High degree of accuracy



Discovering the FLF measurement

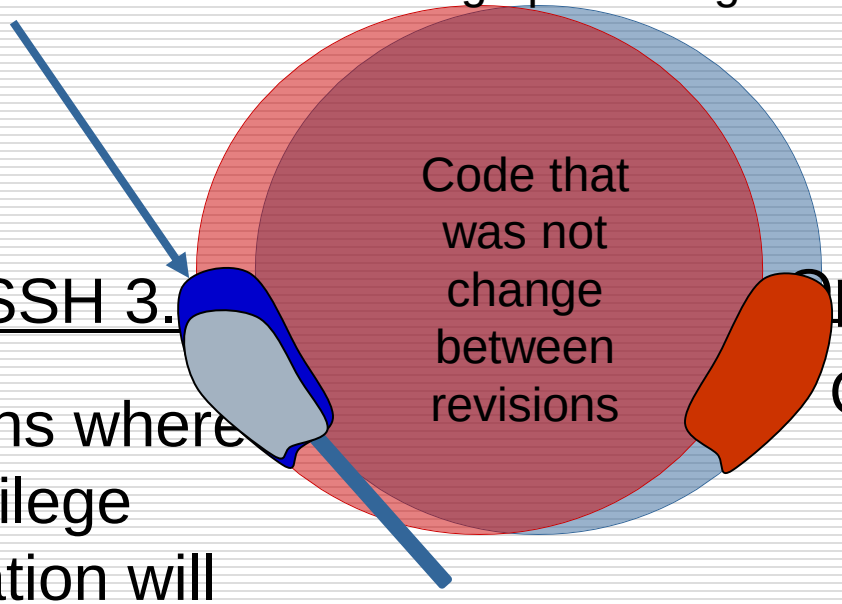
- Collect software systems with known vulnerabilities
- Perform detailed static analyses of software systems
- Calculate areas of likely vulnerability from information gathered during static analyses
- Build tools around these calculations

How are these tools used?

- Run static analysis tools on source code
- A database of code facts is created
- The database is used to find the likely vulnerable functions
- The likely vulnerable functions are outputted and ranked by proximity to input source

Case Study: OpenSSH

GOAL: FLF Finder identifies large percentage of modified functions



OpenSSH 3.

Functions where privilege separation will be implemented

OpenSSH 3.2.3

Code changed to implement privilege separation

82% of changed functions identified by FLF Finder

Experimental Systems

- 30 open source systems
- 31 software security vulnerabilities
- Each system has a single patch file which addresses one security vulnerability
- Most recovered from Redhat's source RPM distribution due to their incremental nature

GAST-MP & SGA

- GNU Abstract Syntax Tree Manipulation Program (GAST-MP)
  - Source Code Analysis tool
  - Operates on G++'s Abstract Syntax Tree (AST)
    - AST can be outputted with the –fdump-tree-flag
  - Creates a repository of code facts
- System Graph Analyzer (SGA)
  - Operates on the code fact repository
  - Identifies Inputs and Targets
  - Performs invocation analysis
  - Calculates FLF Density
  - Analysis of Categorical Graphs

Finding inputs

- An Input is a function which contains reads in external user input
  - For example, `read`
- A list of external function calls were compiled to properly identify Inputs
- This list could be modified to contain application specific library calls

Finding targets

- A Target is any function that contains a known vulnerability
- Targets are found by matching code facts on subtractive lines in a patch file with code facts in the repository generated by GAST-MP

```
--- channels.c      27 Feb 2002 21:23:13 -0000    1.170
+++ channels.c      4 Mar 2002 19:37:58 -0000     1.171
@@ -146,7 +146,7 @@
  {
          Channel *c;

-         if (id < 0 || id > channels_alloc) {
+         if (id < 0 || id >= channels_alloc) {
                  log("channel_lookup: %d: bad id", id);
                  return NULL;
  }
```

FLF density

- Create entire call graph *G*
- Transform *G* in DAG
- Label Input and Target Nodes
- Calculate invocation paths between Input and Target combinations and measure length
- Calculate FLF Density by normalizing path length by function cardinality
- For each system choose the largest FLF Density

Experimental results

System Name	FLF Density (%)	Longest Path	Total Functions	Lines of Code	System Name	FLF Density (%)	Longest Path	Total Functions	Lines of Code
bash	2.18	18	824	67,141	netkit-ping	2.38	1	42	835
crond	2.50	3	120	3,646	netkit-tftpd	1.79	1	56	1,020
elm	1.28	6	468	95,921	nmh	1.53	12	785	52,356
exim	1.82	10	549	61,210	radius-client	4.43	7	158	15,872
fetchmail	3.13	11	351	24,201	screen	.94	4	424	24,796
gnupg	1.16	6	517	73,274	sharutils	6.12	3	49	9,271
inn	.73	3	407	81,429	stunnel	3.52	8	227	3,820
joe	4.04	26	644	20,639	sysklogd	7.58	10	132	6,115
lukemftp	3.04	17	558	7,995	tcpdump	.48	3	627	27,738
lynx	1.00	12	1206	129,420	telnetd	7.14	8	227	16,480
mailx	3.33	10	300	9,351	webalizer	1.33	2	150	6,450
man	3.98	9	226	23,581	wu-ftpd	1.12	4	358	67,755
minicom	3.91	10	256	11,571	wwwoffle	2.85	11	386	44,498
mutt	1.32	15	1139	62,824	zgv-1	3.32	9	271	8,607
netkit-ftp	2.97	7	236	76,695	zgv-2	2.58	7	271	8,607
netkit-inetd	5.50	5	91	1,351					

Experimental results

- Sample mean FLF Density 2.87%
  - Therefore, a very small number of functions are actually likely to be vulnerable
- Standard deviation of 1.87%
  - The FLF density was consistent across our experimental systems
- With 95% confidence the true mean is between 2.23% and 3.51%
  - There is a high probability that our experimental density is close to the TRUE FLF Density

Verification

- The FLF Density can be used as a conservative way to highlight those vulnerability functions which do not have known vulnerabilities

Function Name	Function Coverage (%)	Number Vuln.	Number Found
micq	35.58	3	3
elm	39.82	1	1
dhcpd	18.75	2	1

FLF finder

- FLF Density can say what areas of code are statistically likely to be vulnerable
- Automate tool to find these areas
- Targets are not provided – what do we do?
  - Assume all functions are targets
  - This extremely conservative assumption is still able to reduce 60% of code!

Discussion …

- How would you test for buffer overflow vulnerability?
  - Web application
  - Command line application
  - GUI application
  - Server

You now know …

- … threat modeling
- … malware overview
- … buffer overflow attacks
- … preventing buffer overflow attacks
- … front line functions and their role in security auditing