Notes from Hacking: The Art of Exploitation

By: Programmercave

1. Programming

A. The x86 Processor

1. EAX – Accumulator
2. ECX – Counter
3. EDX – Data ------Registers----------
4. EBX – Base
5. ESP – Stack Pointer
6. EBP – Base Pointer
7. ESI – Source Index --------Registers--------------------
8. EDI – Destination Index
9. EIP – Instruction Pointer
10. EFLAGS register actually consists of several bit flags that are used for comparisons and memory segmentations.

B. Assembly Language

1. The assembly instructions in Intel syntax generally follow this style:

   ```
   operation <destination>, <source>
   ```

2. The destination and source values will either be a register, a memory address, or a value.
3. Formats used with examine (`x`) command in GDB:

   `o` Display in octal.

   `x` Display in hexadecimal.

   `u` Display in unsigned, standard base-10 decimal.

   `t` Display in binary.

4. The memory the EIP register is pointing to can be examined by using the address stored in EIP. The debugger lets you reference registers directly, so $eip is equivalent to the value EIP contains at that moment.
5. The default size of a single unit is a four-byte unit called a ***word***.
6. Valid size letters can used with `x` command:

   `b` A single byte

   `h` A halfword, which is two bytes in size

   `w` A word, which is four bytes in size

   `g` A giant, which is eight bytes in size
7. Sometimes the term ***word*** also refers to 2-byte values. In this case a ***double word*** or ***DWORD*** refers to a 4-byte value
8. x86 processor values are stored in little-endian byte order, which means the least significant byte is stored first.
9. The `lea` instruction is an acronym for ***Load Effective Address***, which will load the familiar address of EBP minus 4 into the EAX register.

C. Back to Basics

a. Pointers

1. In order to see the actual data stored in the pointer variable, you must use the address-of operator (`&`).
2. When it's used, the address of that variable is returned, instead of the variable itself. This operator exists both in GDB and in the C programming language.
3. ***dereference*** operator (`*`)will return the data found in the address the pointer is pointing to, instead of the address itself. This operator exists both in GDB and in the C programming language.

b. Typecasting

1. The syntax for typecasting is as follows:

```
(typecast_data_type) variable
```

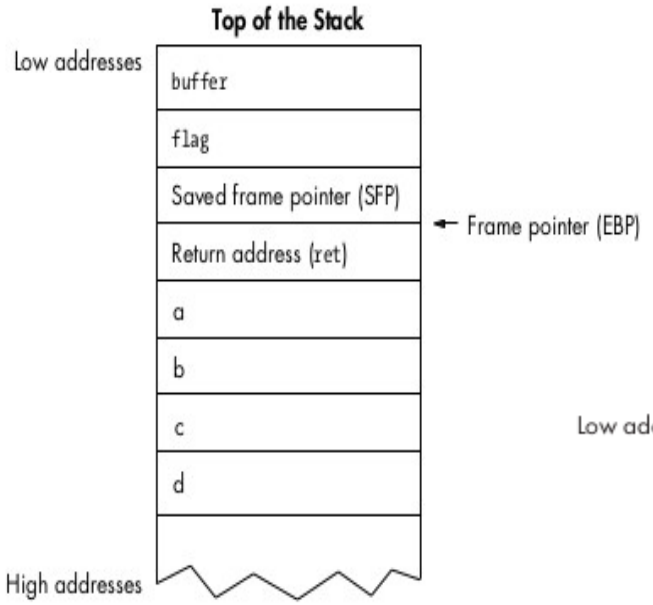
D. Segmentation

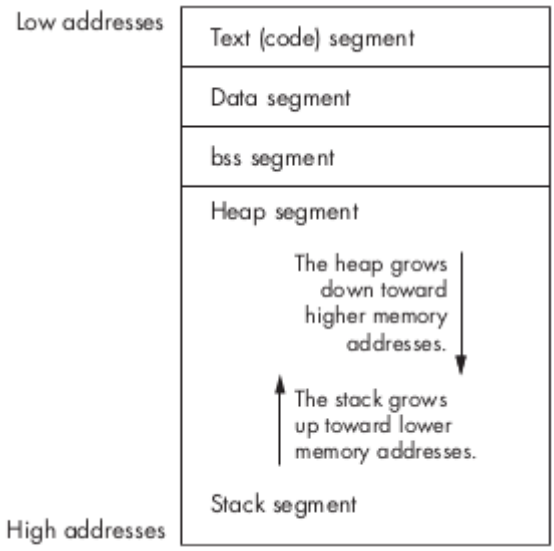
1. A compiled program's memory is divided into five segments: text, data, bss, heap, and stack.

2. The ***text segment*** is also sometimes called the ***code segment***. This is where the assembled machine language instructions of the program are located.
3. Write permission is disabled in the text segment, as it is not used to store variables, only code.
4. Another advantage of this segment being read-only is that it can be shared among different copies of the program, allowing multiple executions of the program at the same time without any problems.
5. The ***data segment*** is filled with the initialized global and static variables, while the ***bss segment*** is filled with their uninitialized counterparts.
6. The ***heap segment*** is a segment of memory a programmer can directly control.
7. All of the memory within the heap is managed by allocator and deallocator algorithms, which respectively reserve a region of memory in the heap for use and remove reservations to allow that portion of memory to be reused for later reservations.
8. The ***stack segment*** also has variable size and is used as a temporary scratch pad to store local function variables and context during function calls.
9. The stack is used to remember all of the passed variables, the location the EIP should return to after the function is finished, and all the local variables used by that function.
10. All of this information is stored together on the stack in what is collectively called a *stack frame*. The stack contains many stack frames.
11. The ESP register is used to keep track of the address of the end of the stack, which is constantly changing as items are pushed into and popped off of it.
12. The EBP register—sometimes called the *frame pointer (FP)* or *local base (LB) pointer*—is used to reference local function variables in the current stack frame.
13. Each stack frame contains the parameters to the function, its local variables, and two pointers that are necessary to put things back the way they were: the *saved frame pointer (SFP)* and the return address.
14. The SFP is used to restore EBP to its previous value, and the return address is used to restore EIP to the next instruction found after the function call.
15. The function arguments are pushed onto the stack in reverse order (since it's FILO).
16.
```
void test_function(int a, int b, int c, int d) {
    int flag;
    char buffer[10];
```

```
    flag = 31337;
    buffer[0] = 'A';
}

int main(){
    test_function(1, 2, 3, 4);
}
```





E. Building on Basics

a. File Access

1. The files fcntl.h and sys/stat.h had to be included, since those files define theflags used with the ***open()*** function.
2. The access mode must use at least one of the following three flags:

   `O_RDONLY` Open file for read-only access.

`O_WRONLY` Open file for write-only access

`O_RDWR` Open file for both read and write access.

A few of the more common and useful of these flags are as follows:

`O_APPEND` Write data at the end of the file.

`O_TRUNC` If the file already exists, truncate the file to 0 length.

`O_CREAT` Create the file if it doesn't exist.

b. File Permissions

1. If the `O_CREAT` flag is used in access mode for the *open()* function, an additional argument is needed to define the file permissions of the newly created file.
2. This argument uses bit flags defined in sys/stat.h, which can be combined with each other using bitwise OR logic.

   `S_IRUSR` Give the file read permission for the user (owner).

   `S_IWUSR` Give the file write permission for the user (owner).

   `S_IXUSR` Give the file execute permission for the user (owner).

   `S_IRGRP` Give the file read permission for the group.

   `S_IWGRP` Give the file write permission for the group.

   `S_IXGRP` Give the file execute permission for the group.

   `S_IROTH` Give the file read permission for other (anyone).

   `S_IWOTH` Give the file write permission for other (anyone).

   `S_IXOTH` Give the file execute permission for other (anyone).
3. Unix file permissions

   First, the user read/write/execute permissions are displayed, using r for read, w for write, x for execute, and - for off.

   The next three characters display the group permissions, and the last three characters are for the other permissions.

   Each permission corresponds to a bit flag; read is 4 (100 in binary), write is 2 (010 in binary), and execute is 1 (001 in binary).

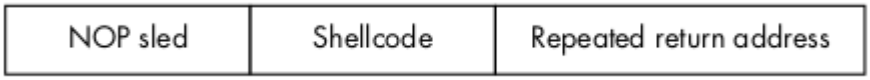
   The argument `ugo-wx` means *Subtract write and execute permissions from user, group, and other.*

   `chmod u+w` command gives write permission to user.

2. Exploitation

A. Stack-Based Buffer Overflow Vulnerabilities

1. NOP is an assembly instruction that is short for *no operation*.
2. It is a single-byte instruction that does absolutely nothing.
3. If the EIP register points to any address found in the NOP sled, it will increment while executing each NOP instruction, one at a time, until it finally reaches the shellcode.
4. NOP instruction is equivalent to the hex byte 0x90.



5. In situations where the overflow buffer isn't large enough to hold shellcode, an environment variable can be used with a large NOP sled. This usually makes exploitations quite a bit easier.
6. In C's standard library there is a function called `getenv()`, which accepts the name of an environment variable as its only argument and returns that variable's memory address.
7. The length of the name of the executing program has an effect on the location of exported environment variables.
8. The general trend seems to be a decrease of two bytes in the address of the environment variable for every single-byte increase in the length of the program name.
9. This must mean the name of the executing program is also located on the stack somewhere, which is causing the shifting.
10. The system() function is used to execute a command.
11. This function starts a new process and runs the command using `/bin/sh -c`.
12. The `-c` tells the sh program to execute commands from the command-line argument passed to it.
13. The `execl()` function belongs to a family of functions that execute commands by replacing the current process with the new one.
14. The arguments for `execl()` start with the path to the target program and are followed by each of the command-line arguments.

15. The `execl()` function has a sister function called `execle()` , which has one additional argument to specify the environment under which the executing process should run.

B. A Basic Heap-Based Overflow

1. `/etc/passwd` file contains all of the usernames, IDs, and login shells for all the users of the system.
2. The fields in the `/etc/passwd` file are delimited by colons, the first field being for login name, then password, user ID, group ID, username, home directory, and finally the login shell.
3. The password fields are all filled with the x character, since the encrypted passwords are stored elsewhere in a shadow file.
4. Any entry in the password file that has a user ID of 0 will be given root privileges.
5. The password can be encrypted using a one-way hashing algorithm.
6. To prevent lookup attacks, the algorithm uses a *salt value*, which when varied creates a different hash value for the same input password.
7. Salt value is always at the beginning of the hash.
8. Using the salt value from the stored encrypted password, the system uses the same one-way hashing algorithm to encrypt whatever text the user typed as the password. Finally, the system compares the two hashes; if they are the same, the user must have entered the correct password.
9. The `nm` command lists symbols in object files. This can be used to findaddresses of various functions in a program.

```
nm <object_file>
```

C. Format Strings

1. When a format function encounters a `%n` format parameter, it writes the number of bytes that have been written by the function to the address in the corresponding function argument.
2. The unary address operator (`&`) is used to write this data.
3. `printf("A is %d and is at %08x. B is %x.\n", A, &A, B);`

   When this `printf()` function is called (as with any function), the argu- ments are pushed to the stack in reverse order. First the value of B , then the address of A , then the value of A , and finally the address of the format string.
4. `reader@hacking:~/booksrc $ ./fmt_vuln testing%x`

When the %x format parameter was used, the hexadecimal representation of a four-byte word in the stack was printed. This process can be used repeatedly to examine stack memory.

5. The %s format parameter can be used to read from arbitrary memory addresses.

```
reader@hacking:~/booksrc $ env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0xbffffdd7
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/
usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
```

6. The %n format parameter can be used to write to arbitrary memory addresses.
7. By manipulating the field-width option of one of the format parameters before the %n , a certain number of blank spaces can be inserted, resulting in the output having some blank lines.
8. These lines, in turn, can be used to control the number of bytes written before the %n format parameter.
9. ***Direct Parameter Access*** allows parameters to be accessed directly by using the dollar sign qualifier.
10. For example, %n$d would access the nth parameter and display it as a decimal number.
11. A function can be declared as a destructor function by defining the destructor attribute
12. The behavior of automatically executing a function on exit is controlled by the .dtors table section of the binary.
13. This section is an array of 32-bit addresses terminated by a NULL address. The array always begins with 0xffffffff and ends with the NULL address of 0x00000000 .
14. Between these two are the addresses of all the functions that have been declared with the destructor attribute.
15. .dtors section is writable

16. It is included in all binaries compiled with the GNU C compiler, regardless of whether any functions were declared with the destructor attribute.
17. ***Procedure Linkage Table (PLT)*** section consists of many jump instructions, each one corresponding to the address of a function. It works like a springboard—each time a shared function needs to be called, control will pass through the PLT.
18. Advantage of overwriting the GOT is that the GOT entries are fixed per binary, so a different system with the same binary will have the same GOT entry at the same address. (Global Offset Table)

---

3. Networking

A. OSI Model

1. The OSI model provides standards that allow hardware, such as routers and firewalls, to focus on one particular aspect of communication that applies to them and ignore others.
2. **Physical layer** - primary role is communicating raw bit streams.

   - responsible for activating, maintaining, and deactivating these bit-stream communications.
3. **Data-link layer –** error correction and flow control.

   - provides procedures for activating, maintaining, and deactivating data-link connections.
4. **Network layer** - pass information between the lower and the higher layers.

   - provides addressing and routing.
5. **Transport layer** - transparent and reliable transfer of data between systems.
6. **Session layer** - responsible for establishing and maintaining connections between network applications.
7. **Presentation layer** - responsible for presenting the data to applications in a syntax or language they understand.

   - allows for things like encryption and data compression.
8. **Application layer** - keeping track of the requirements of the application.
9. Each wrapped layer contains a header and a body.

B. Sockets

1. To the programmer, a socket can be used to send or receive data over a network.
2. ***Stream sockets*** provide reliable two-way communication similar to when you call someone on the phone.
3. Stream sockets use a standard communication protocol called Transmission Control Protocol (TCP), which exists on the transport layer (4) of the OSI model.
4. Communicating with a ***datagram socket*** is more like mailing a letter than making a phone call.
5. The connection is one-way only and unreliable.
6. Datagram sockets use another standard protocol called UDP instead of TCP on the transport layer (4). (User Datagram Protocol)
7. Socket functions have their prototypes defined in `/usr/include/sys/sockets.h`.

C. A Web Client Example

1. By default, webservers run on port 80, which is listed along with many other default ports in /etc/services.
2. In the language of HTTP, requests are made using the command `GET` , followed by the resource path and the HTTP protocol version.
3. For example, `GET / HTTP/1.0` will request the root document from the webserver using HTTP version 1.0.
4. If the server finds the resource, it will respond using HTTP by sending several headers before sending the content.
5. If the command `HEAD` is used instead of `GET` , it will only return the HTTP headers without the content.
6. These headers can be retrieved manually using telnet by connecting to port 80 of a known website, then typing `HEAD / HTTP/1.0` and pressing ENTER twice.

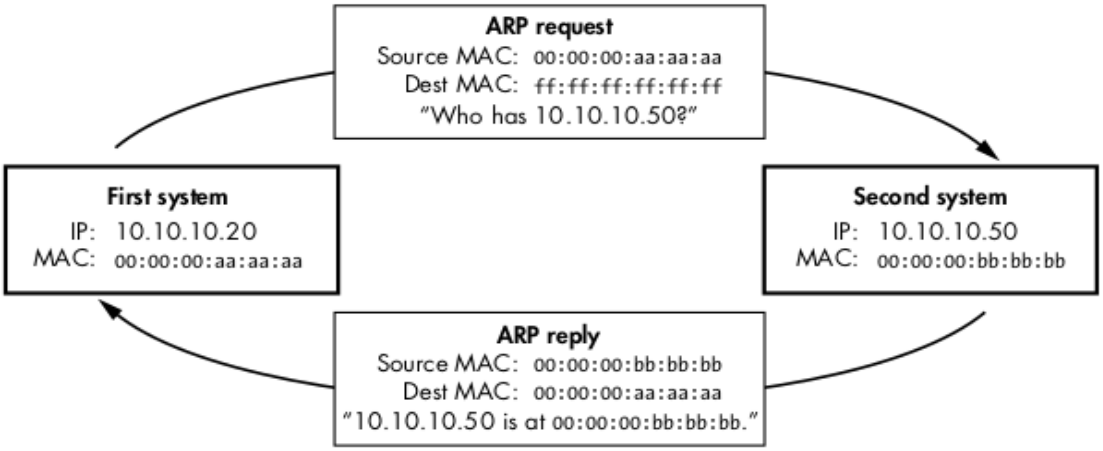
   - `$ telnet www.internic.net 80`
7. DNS is a protocol that allows an IP address to be looked up by a named address, similar to how a phone number can be looked up in a phone book if you know the name.

D. Peeling Back the Lower Layers

1. Sockets exist on the session layer (5), providing an interface to send data from one host to another.
2. TCP on the transport layer (4) provides reliability and transport control.
3. IP on the network layer (3) provides addressing and packet-level communication.
4. Ethernet on the data-link layer (2) provides addressing between Ethernet ports, suitable for basic LAN (Local Area Network) communications.
5. Physical layer (1) is simply the wire and the protocol used to send bits from one device to another.

a. Data-Link Layer

1. Ethernet exists on this layer, providing a standard addressing system or all Ethernet devices.
2. These addresses are known as Media Access Control (MAC) addresses.
3. Consisting of six bytes, usually written in hexadecimal in the form `xx:xx:xx:xx:xx:xx`.
4. Stored in the device's integrated circuit memory.
5. Ethernet header is 14 bytes in size and contains the source and destination MAC addresses for this Ethernet packet.
6. Ethernet addressing also provides a special broadcast address, consisting of all binary 1's (`ff:ff:ff:ff:ff:ff`).
7. Any Ethernet packet sent to this address will be sent to all the connected devices.
8. Address Resolution Protocol (ARP) protocol allows "seating charts" to be made to associate an IP address with a piece of hardware.
9. An ARP request is a message, sent to the broadcast address, that contains the sender's IP address and MAC address and basically says, "Hey, who has this IP? If it's you, please respond and tell me your MAC address."
10. An ARP reply is the corresponding response that is sent to the requester's MAC address (and IP address) saying, "This is my MAC address, and I have this IP address."

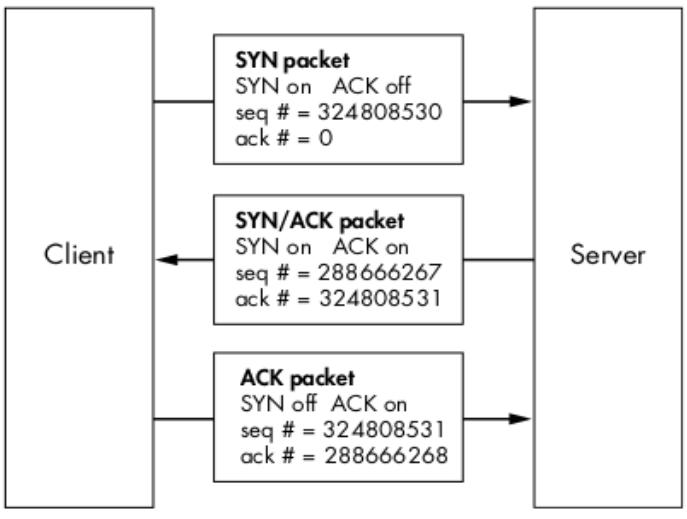


b. Network Layer

1. Every system on the Internet has an IP address, consisting of a familiar four-byte arrangement in the form of `xx.xx.xx.xx`.
2. The header carries a checksum, to help detect transmission errors, and fields to deal with packet fragmentation.
3. There's no guarantee that an IP packet will actually reach its final destination. If there's a problem, an ICMP packetis sent back to notify the sender of the problem.
4. ICMP Echo Request and Echo Reply messages are used by a utility called ping.
5. ICMP and IP are both connectionless; all this protocol layer really cares about is getting the packet to its destination address.

c. Transport Layer

1. TCP is the most commonly used protocol for services on the Internet: telnet, HTTP (webtraffic), SMTP (email traffic), and FTP (file transfers) all use TCP.
2. Provides a transparent, yet reliable and bidirectional, connection between two IP addresses.
3. Stream sockets use TCP/IP connections.
4. The SYN and ACK flags are used together to open connections in a three-step handshaking process.



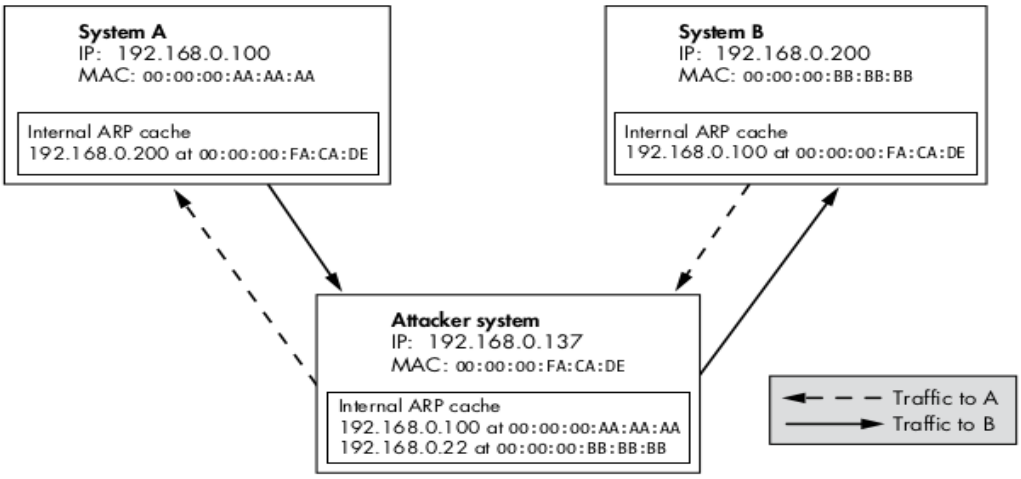
5. Sequence numbers allow TCP to put unordered packets back into order, to determine whether packets are missing, and to prevent mixing up packets from other connections.
6. When a connection is initiated, each side generates an initial sequence number. This number is communicated to the other side in the first two SYN packets of the connection handshake.
7. Then, with each packet that is sent, the sequence number is incremented by the number of bytes found in thedata portion of the packet. This sequence number is included in the TCP packet header.
8. Each TCP header has an acknowledgment number, which is simply the other side's sequence number plus one.

E. Network Sniffing

1. On an *unswitched network*, Ethernet packets pass through every device on the network, expecting each system device to only look at the packets sent to its destination address.
2. It's fairly trivial to set a device to *promiscuous mode*, which causes it to look at all packets, regardless of the destination address.
3. Promiscuous mode can be set using `ifconfig`.

   ```
   $ sudo ifconfig eth0 promisc
   ```

4. The act of capturing packets that aren't necessarily meant for public viewing is called ***sniffing***.
5. The act of forging a source address in a packet is known as ***spoofing***.
6. Spoofing is the first step in sniffing packets on a switched network.
7. The attacker sends spoofed ARP replies to certain devices that cause the ARP cache entries to be overwritten with the attacker's data. This technique is called ARP ***cache poisoning***.
8. In order to sniff network traffic between two points, A and B, the attacker needs to poison the ARP cache of A to cause A to believe that B's IP address is at the attacker's MAC address, and also poison the ARP cache of B to cause B to believe that A's IP address is also at the attacker's MAC address.



9. A

gateway is a system that routes all the traffic from a local network out tothe Internet.

F. Denial of Service

a. SYN Flooding

1. The attacker floods the victim's system with many SYN packets, using a spoofed nonexistent source address.
2. Since a SYN packet is used to initiate a TCP connection, the victim's machine will send a SYN/ACK packet to the spoofed address in response and wait for the expected ACK response.
3. Each of these waiting, half-open connections goes into a backlog queue that has limited space.
4. As long as the attacker continues to flood the victim's system with spoofed SYN packets, the victim's backlog queue will remain full, making it nearly impossible for real SYN packets to get to the system and initiate valid TCP/IP connections.

b. The Ping of Death

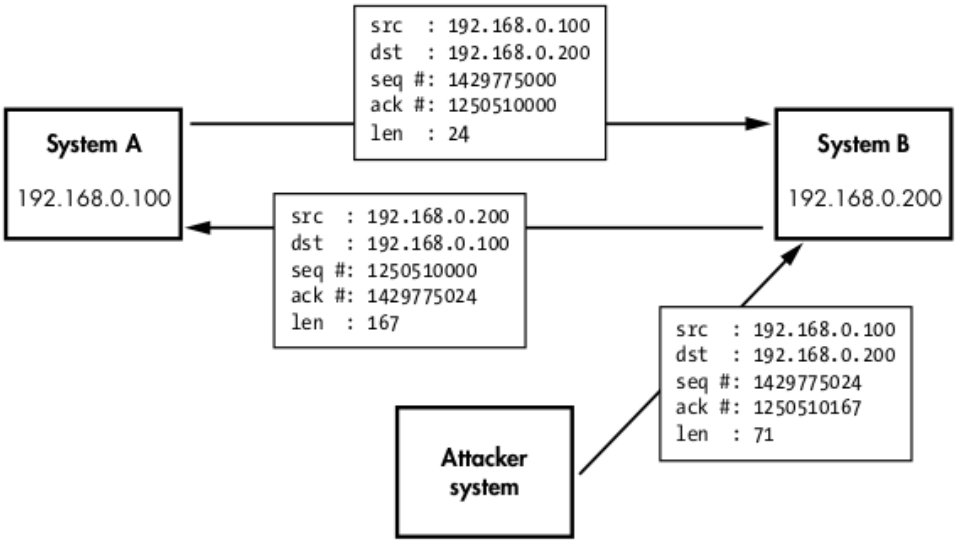
1. According to the specification for ICMP, ICMP echo messages can only have 2^{16} , or 65,536, bytes of data in the data part of the packet.
2. Several operating systems crashed if they were sent ICMP echo messages that exceeded the size specified.

c. Teardrop

1. When a packet is fragmented, the offsets stored in the header will line up to reconstruct the original packet with no overlap.
2. The teardrop attack sent packet fragments with overlapping offsets, which caused implementations that didn't check for this irregular condition to inevitably crash.

G. TCP/IP Hijacking

1. To carry out a TCP/IP hijacking attack, the attacker must be on the same network as the victim.
2. By sniffing the local network segment, all of the details of open TCP connections can be pulled from the headers.
3. While sniffing, the attacker has access to the sequence numbers for a connection between a victim (system A) and a host machine (system B).
4. Then the attacker sends a spoofed packet from the victim's IP address to the host machine, using the sniffed sequence number to provide the proper acknowledgment number.



a. RST Hijacking

1. A very simple form of TCP/IP hijacking involves injecting an authentic-lookingreset (RST) packet.
2. If the source is spoofed and the acknowledgment number is correct, the receiving side will believe that the source actually sent the reset packet, and the connection will be reset.

H. Port Scanning

1. Port scanning is a way of figuring out which ports are listening and accepting connections.

a. Stealth SYN Scan

1. A SYN scan is also sometimes called a half-open scan. This is because it doesn't actually open a full TCP connection.
2. Only the initial SYN packet is sent, and the response is examined.

3. If a SYN/ACK packet is received in response, that port must be accepting connections.This is recorded, and an RST packet is sent to tear down the connection to prevent the service from accidentally being DoSed.
4. Using nmap, a SYN scan can be performed using the command-line option `-sS`.

b. FIN, X-mas and Null Scans

1. FIN, X-mas, and Null scans all involve sending a nonsensical packet to every port on the target system.
2. If a port is listening, these packets just get ignored. However, if the port is closed and the implementation follows protocol (RFC 793), an RST packet will be sent.
3. This difference can be used to detect which ports are accepting connections, without actually opening any connections.
4. The FIN scan sends a FIN packet, the X-mas scan sends a packet with FIN, URG, and PUSH turned on (so named because the flags are lit up like a Christmas tree), and the Null scan sends a packet with no TCP flags set.
5. Using nmap, FIN, X-mas, and NULL scans can be performed using the command-line options `-sF , -sX , and -sN`, respectively.

c. Spoofing Decoys

1. This technique simply spoofs connections from various decoy IP addresses in between each real port-scanning connection.
2. The spoofed decoy addresses must use real IP addresses of live hosts; otherwise, the target may be accidentally SYN flooded.
3. The sample nmap command shown below scans the IP 192.168.42.72, using 192.168.42.10 and 192.168.42.11 as decoys.

```
$ sudo nmap -D 192.168.42.10,192.168.42.11 192.168.42.72
```

d. Idle Scanning

1. The attacker needs to find a usable idle host that is not sending or receiving any other network traffic and that has a TCP implementation that produces predictable IP IDs that change by a known increment with each packet.
2. IP IDs are meant to be unique per packet per session, and they are commonly incremented by a fixed amount.
3. First, the attacker gets the current IP ID of the idle host by contacting it with a SYN packet or an unsolicited SYN/ACK packet and observing the IP ID of the response.
4. By repeating this process a few more times, the increment applied to the IP ID with each packet can be determined.

5. Then, the attacker sends a spoofed SYN packet with the idle host's IP address to a port on the target machine.
6. If that port is listening, a SYN/ACK packet will be sent back to the idle host. But since the idle host didn't actually send out the initial SYN packet, this response appears to be unsolicited to the idle host, and it responds by sending back an RST packet.
7. If that port isn't listening, the target machine doesn't send a SYN/ACK packet back to the idle host, so the idle host doesn't respond.
8. At this point, the attacker contacts the idle host again to determine how much the IP ID has incremented.
9. If it has only incremented by one interval, no other packets were sent out by the idle host between the two checks. This implies that the port on the target machine is closed.
10. If the IP ID has incremented by two intervals, one packet, presumably an RST packet, was sent out by the idle machine between the checks. This implies that the port on the target machine is open.
11. This type of scanning can be done with nmap using the `-sI` command-line option followed by the idle host's address.

---

4. Shellcode

A. Assembly vs. C

1. The shellcode bytes are actually architecture-specific machine instructions, so shellcode is written using the assembly language.
2. A C program compiled on an x86 processor will produce x86 assembly language.
3. By definition, assembly language is already specific to a certain processor architecture, so portability is impossible.
4. File descriptors are used for almost everything in Unix: input, output, file access, network sockets, and so on.

   0 for Standard Input, 1 for Standard Output and 2 for Standard error Output

a. Linux System Calls in Assembly

1. Every possible Linux system call is enumerated, so they can be referenced by numbers when making the calls in assembly.
2. These syscalls are listed in /usr/include/asm-i386/unistd.h.

3. Assembly instructions for the x86 processor have one, two, three, or no operands.
4. The operands to an instruction can be numerical values, memory addresses, or processor registers.
5. The x86 processor has several 32-bit registers that can be viewed as hardware variables.
6. The registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP can all be used as operands, while the EIP register (execution pointer) cannot.
7. The `mov` instruction copies a value between its two operands.
8. The `int` instruction sends an interrupt signal to the kernel, defined by its single operand.
9. With the Linux kernel, interrupt `0x80` is used to tell the kernel to make a system call.
10. The EAX register is used to specify which system call to make, while the EBX, ECX, and EDX registers are used to hold the first, second, and third arguments to the system call.
11. The string "Hello, world!" with a newline character (0x0a) is in the data segment, and the actual assembly instructions are in the text segment.
12. The nasm assembler with the `-f elf` argument will assemble the helloworld.asm into an object file ready to be linked as an ELF binary.

B. The Path to Shellcode

1. In shellcode, the bytes for the string "Hello, world!" must be mixed together with the bytes for the assembly instructions, since there aren't definable or predictable memory segments.
2. To access the string as data we need a pointer to it.
3. When the shellcode gets executed, it could be anywhere in memory. The string's absolute memory address needs to be calculated relative to EIP.

a. Assembly Instructions Using the Stack

1. `push <source>` - Push the source operand to the stack.
2. `pop <destination>` - Pop a value from the stack and store in the destination operand.
3. `call <location>` - Call a function, jumping the execution to the address in the location operand. This location can be relative or absolute. The address of the

instruction following the call is pushed to the stack, so that execution can return later.

4. `ret` - Return from a function, popping the return address from the stack and umping execution there.
5. By overwriting the stored return address on the stack before the `ret` instruction, we can take control of a program's execution.
6. The nasm assembler converts assembly language into machine code and a corresponding tool called ndisasm converts machine code into assembly.

C. Shell-Spawning Shellcode

1. The `lea` instruction, whose nam stands for *load effective address*, works like the `address-of` operator in C.

   ```
   lea <dest>, <source> - Load the effective address of the source operand into the destination operand.
   ```

2. With Intel assembly syntax, operands can be dereferenced as pointers if they are surrounded by square brackets.
3. The following instruction in assembly will treat EBX+12 as a pointer and write eax to where it's pointing.

   ```
   89 43 0C            mov [ebx+12],eax
   ```

---

5. Countermeasures

1. A daemon is a program that runs in the background and detaches from the controlling terminal in a certain way.
2. Daemon programs typically end with a *d* to signify they are daemons, such as *sshd* or *syslogd*.

A. Crash Course in Signals

1. When a process receives a signal, its flow of execution is interrupted by the operating system to call a signal handler.
2. Signals are identified by a number, and each one has a default signal handler.
3. Custom signal handlers can be registered using the `signal()` function.
4. Specific signals can be sent to a process using the `kill` command.
5. By default, the `kill` command sends the terminate signal (SIGTERM) to a process.

6. With the `-l` command-line switch, `kill` lists all the possible signals.

B. Payload smuggling

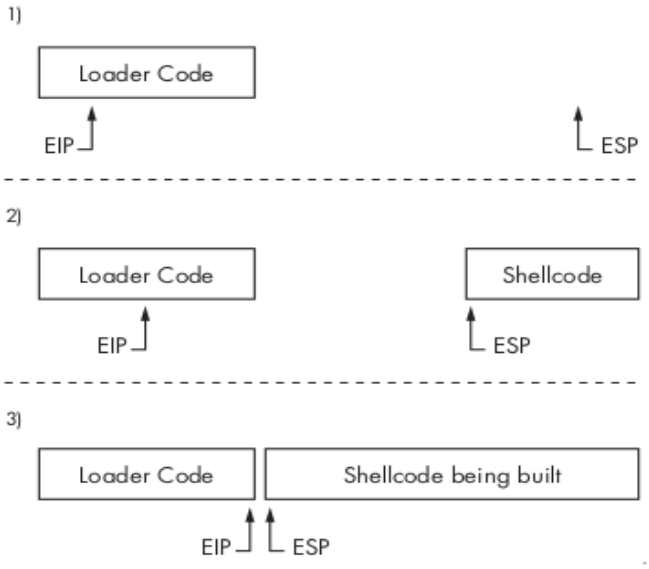
a. String Encoding

1. To hide the string, we will simply add 5 to each byte in the string. Then, after the string has been pushed to the stack, the shellcode will subtract 5 from each string byte on the stack.

C. Buffer Restrictions

a. Polymorphic Printable ASCII Shellcode

1. Polymorphic shellcode refers to any shellcode that changes itself.
2. The goal is to write shellcode that will get past the printable character check.
3. XOR instruction on the various registers doesn't assemble into the printable ASCII character range.
4. AND bitwise operation assembles into the percent character (%) when using the EAX register.
5. The assembly instruction of `and eax, 0x41414141` will assemble to the printable machine code of %AAAA.
6. If two inverse values are ANDed onto EAX, EAX will become zero.
7. `%JONE%501:` in machine code will zero out the EAX register.
8. The general technique is, first, to set ESP back behind the executing loader code (in higher memory addresses), and then to build the shellcode from end to start by pushing values onto the stack.
9. Since the stack grows up (from higher memory addresses to lower memory addresses), the ESP will move backward as values are pushed to the stack, and the EIP will move forward as the loader code executes. Eventually, EIP and ESP will meet up, and the EIP will continue executing into the freshly built shellcode.



D. Nonexecutable Stack

1. Obvious defense against buffer overflow exploits is to make the stack nonexecutable.
2. When this is done, shellcode inserted anywhere on the stack is basically useless.

a. ret2libc

1. There exists a technique used to bypass this protective countermeasure known as *returning into libc*. libc is a standard C library that contains various basic functions, such as `printf()` and `exit()`.
2. Any program that uses the `printf()` function directs execution into the appropriate location in libc.
3. An exploit can do the exact same thing and direct a program's execution into a certain function in libc.

b. Returning into system()

1. One of the simplest libc functions to return into is `system()`.
2. This function takes a single argument and executes that argument with /bin/sh.
3. First, the location of the `system()` function in libc must be determined.
4. This will be different for every system, but once the location is known, it will remain the same until libc is recompiled.

E. Randomized Stack Space

a. Bouncing Off linux-gate

1. Bouncing off linux-gate refers to a shared object, exposed by the kernel, which looks like a shared library.
2. The important thing is that every process has a block of memory containing linux-gate's instructions, which are always at the same location, even with ASLR.

6. Cryptology

A. Information Theroy

a. One-Time Pads

1. A one-time pad is a very simple cryptosystem that uses blocks of random data called *pads*.
2. The pad must be at least as long as the plaintext message that is to be encoded, and the random data on the pad must be truly random.
3. Two identical pads are made: one for the recipient and one for the sender.
4. To encode a message, the sender simply XORs each bit of the plaintext message with the corresponding bit of the pad. After the message is encoded, the pad is destroyed to ensure that it is only used once.
5. When the recipient receives the encrypted message, he also XORs each bit of the encrypted message with the corresponding bit of his pad to produce the original plaintext message.
6. The security of the one-time pad hinges on the security of the pads.

b. Quantum Key Distribution

1. This is done using nonorthogonal quantum states in photons.
2. *Nonorthogonal* simply means the states are separated by an angle that isn't 90 degrees.
3. First, the sender and receiver agree on bit representation for the four polarizations, such that each basis has both 1 and 0.
4. Then, the sender sends a stream of random photons, each coming from a randomly chosen basis (either rectilinear or diagonal), and these photons are recorded.
5. When the receiver receives a photon, he also randomly chooses to measure it in either the rectilinear basis or the diagonal basis and records the result.
6. Now, the two parties publicly compare which basis they used for each photon, and they keep only the data corresponding to the photons they both measured using the same basis.

c. Computational Security

1. A cryptosystem is considered to be *computationally secure* if the best-known algorithm for breaking it requires an unreasonable amount of computational resources and time.

B. Algorithmic Run Time

1. The growth rate of the time complexity of an algorithm with respect to input size is more important than the time complexity for any fixed input.

a. Asymptotic Notation

1. *Asymptotic notation* is a way to express an algorithm's efficiency. It's called asymptotic because it deals with the behavior of the algorithm as the input size approaches the asymptotic limit of infinity.

C. Symmetric Encryption

1. *Symmetric ciphers* are cryptosystems that use the same key to encrypt and decrypt messages.
2. A *block cipher* operates on blocks of a fixed size, usually 64 or 128 bits.
3. The same block of plaintext will always encrypt to the same ciphertext block, using the same key.
4. DES, Blowfish, and AES (Rijndael) are all block ciphers.
5. *Stream ciphers* generate a stream of pseudo-random bits, usually either one bit or byte at a time. This is called the *keystream*, and it is XORed with the plaintext.
6. This is useful for encrypting continuous streams of data.
7. RC4 and LSFR are examples of popular stream ciphers.
8. Two concepts used repeatedly in block ciphers are confusion and diffusion.
9. *Confusion* refers to methods used to hide relationships between the plaintext, the ciphertext, and the key.
10. Diffusion serves to spread the influence of the plaintext bits and the key bits over as much of the ciphertext as possible.

a. Lov Grover's Quantum Search Algorithm

1. A quantum computer can store many different states in a superposition (which can be thought of as an array) and perform calculations on all of them at once.
2. This is ideal for brute forcing anything, including block ciphers.
3. The superposition can be loaded up with every possible key, and then the encryption operation can be performed on all the keys at the same time.
4. The tricky part is getting the right value out of the superposition.
5. Quantum computers are weird in that when the superposition is looked at, the whole thing decoheres into a single state.

6. Fortuitously, a man named Lov Grover came up with an algorithm that can manipulate the odds of the superposition states.
7. This algorithm allows the odds of a certain desired state to increase while the others decrease.
8. This process is repeated several times until the decohering of the superposition into the desired state is nearly guaranteed. This takes about O $n^{½}$ steps.

D. Asymmetric Encryption

1. Asymmetric ciphers use two keys: a public key and a private key.
2. Any message that is encrypted with the public key can only be decrypted with the private key.
3. Unlike symmetric ciphers, there's no need for an out-of-band communication channel to transmit the secret key.
4. However, asymmetric ciphers tend to be quite a bit slower than symmetric ciphers.

E. Hybrid Ciphers

1. An asymmetric cipher is used to exchange a randomly generated key that is used to encrypt the remaining communications with a symmetric cipher.
2. This provides the speed and efficiency of a symmetric cipher, while solving the dilemma of secure key exchange.
3. Hybrid ciphers are used by most modern cryptographic applications, such as SSL, SSH, and PGP.

a. Man-in-the-Middle-Attacks

1. The attacker sits between the two communicating parties, with each party believing they are communicating with the other party, but both are communicating with the attacker.

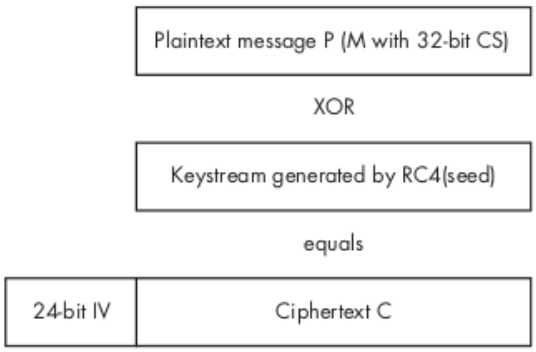
F. Wireless 802.11b Encryption

1. If the wireless network isn't VLANed off or firewalled, an attacker associated to the wireless access point could redirect all the wired network traffic out over the wireless via ARP redirection.

a. Wired Equivalent Privacy

1. WEP was meant to be an encryption method providing security equivalent to a wired access point.

2. All of the encryption is done on a per-packet basis, so each packet is essentially a separate plaintext message to send. The packet will be called M.
3. First, a checksum of message M is computed, so the message integrity can be checked later.
4. This checksum will be called CS, so CS = CRC32(M).
5. This value is appended to the end of the message, which makes up the plaintext message P.
6. Now, the plaintext message needs to be encrypted. This is done using RC4, which is a stream cipher.
7. Then the seed value S is fed into RC4, which will generate a keystream.
8. WEP uses an initialization vector (IV) for the seed value.
9. This keystream is XORed with the plaintext message P to produce the ciphertext C.
10. The IV is prepended to the ciphertext, and the whole thing is encapsulated with yet another header and sent out over the radio link.



11. When the recipient receives a WEP-encrypted packet, the process is simply reversed.

Reference:

Hacking : The Art of Exploitation – Jon Erickson

https://programmercave0.github.io/

https://www.facebook.com/programmercave/

https://t.me/programmercave