Topic 11: Software Security

Explain malware types, software vulnerabilities, their exploitations and countermeasures

Main textbook: Computer Security, 4th Edition by William Stallings and Lawrie Brown, ISBN-13: 9780134794105, Pearson, 2018.

Also a useful book: Computer Security, 2nd Edition by W. Du.

Acknowledgements:

Some of the slides are from the authors of the main text book.

Some of the diagrams are from Computer Security, 2nd Edition by W. Du.

Overview

- Part 1
  - Malware (Malicious Software)
- Part 2
  - Buffer Overflows: Attacks and Defence
  - Conclusion

Main textbook, Chapters 6, 10, 11.
NIST Special Publication 800-83, "Guide to Malware Incident Prevention and Handling for Desktops and Laptops",
https://csrc.nist.gov/publications/detail/sp/800-83/rev-1/final
or
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-83r1.pdf
(as with all the web linked publications, the web address may change, so pls use the article title to search for the doc if you have problem with the link.)

Part 1 - Overview

- Malware (Malicious Software)
  - What is
  - Malware Types
  - Advanced Persistent Threats (APTs)
  - Defence Measures

What is Malware

NIST 800-83 defines malware as:

"a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim."

Malware Types

- May be classified based on
  - How it spreads or propagates to reach its targets, e.g. by exploiting software vulnerabilities, social engineering.
  - Payloads it performs once a target is reached, e.g. corruption of system or files, theft of services or data, stealthing/hiding its presence on the system.

- May also be classified in terms of
  - Whether it needs a host program (e.g. viruses, trapdoors, logic bombs), or a self-contained program (worms, zombie/bot).
  - Whether it is non-replicating (trojans, trapdoors, logic bombs), or self-replicating (viruses, worms, bots).

Malware Types

- There are many types, e.g.

Viruses
Worms
Trapdoor
Trojan horses
Malicious mobile code
Zombies
Ransomware
Keyloggers
Malicious web browser plug-Ins
Adware
Spyware
Attacker toolkits

Viruses & Classifications

- A piece of code that, when executed, inserts itself into a host/target program or data file.
- They can do anything that the host program is permitted to do.
- Viruses are often triggered through user interaction.
- Spread when infected files are moved from machine to machine.
- Specific to operating system and hardware
  - Takes advantage of their details and weaknesses

- Classification by concealment strategy
  - Polymorphic virus: change its appearance, control flow, etc, with each infection.
  - Metamorphic virus: transform the virus code as they propagate in terms of both appearance and behaviour.

Virus - Classifications

- Classification by target
  - Boot sector viruses: infect control sectors on a disk.
  - File infector viruses: infect executable files, e.g. .com, .exe, .bat
  - Multipartite viruses: viruses combine the characteristics of file infector and boot sector viruses.
  - Macro viruses: use the capabilities of macro programming language to infect application documents and document templates.
  - Scripting viruses: infect scripts that are understood by scripting languages.

Virus - Structure

- Components
  - Propagation: need human intervention, e.g. spam emails, unofficial download sources, fake updates.
  - Trigger: event that makes payload execute.
  - Payload: virus function, e.g. ransomware.
- An infected program, upon invocation, first executes virus code and then original program code.

Structure of a basic virus

```
Virus() {
  infectExecutable();
  if (triggered()) {
     doDamage();
  }
  jump to main of infected program;
}

void infectExecutable() {
  file = choose an uninfected executable
file;
  prepend/postpend/embed V to file;
}

void doDamage() { ... }
int triggered() { return (some test? 1 : 0); }
```

Worms

- Active code that can replicate and propagate autonomously over network connections and/or through shared media.
- Different from viruses, a worm doesn't require a host program or human interaction to spread; it usually spreads through a software vulnerability or a vulnerability in a network service.

- Components
  - Target discovery
  - Propagation
  - Activation
  - Payload

Worms - Target Discovery

- Port scanning
  - Search for other systems to infect, e.g. by using port scanning in a sequential or a random order.
- External hit-list
  - Compile a list of potential vulnerable machines, e.g. through external meta-servers.
- Internal topological information
  - Use information contained on an infected victim machine to find more target hosts, e.g. email addresses, URLs on disk and in cache.
- Local subnet address structure
  - If a host can be infected behind a firewall, that host then looks for targets in its own local network, e.g. by using the subnet address structure to find other hosts on the same subnet.

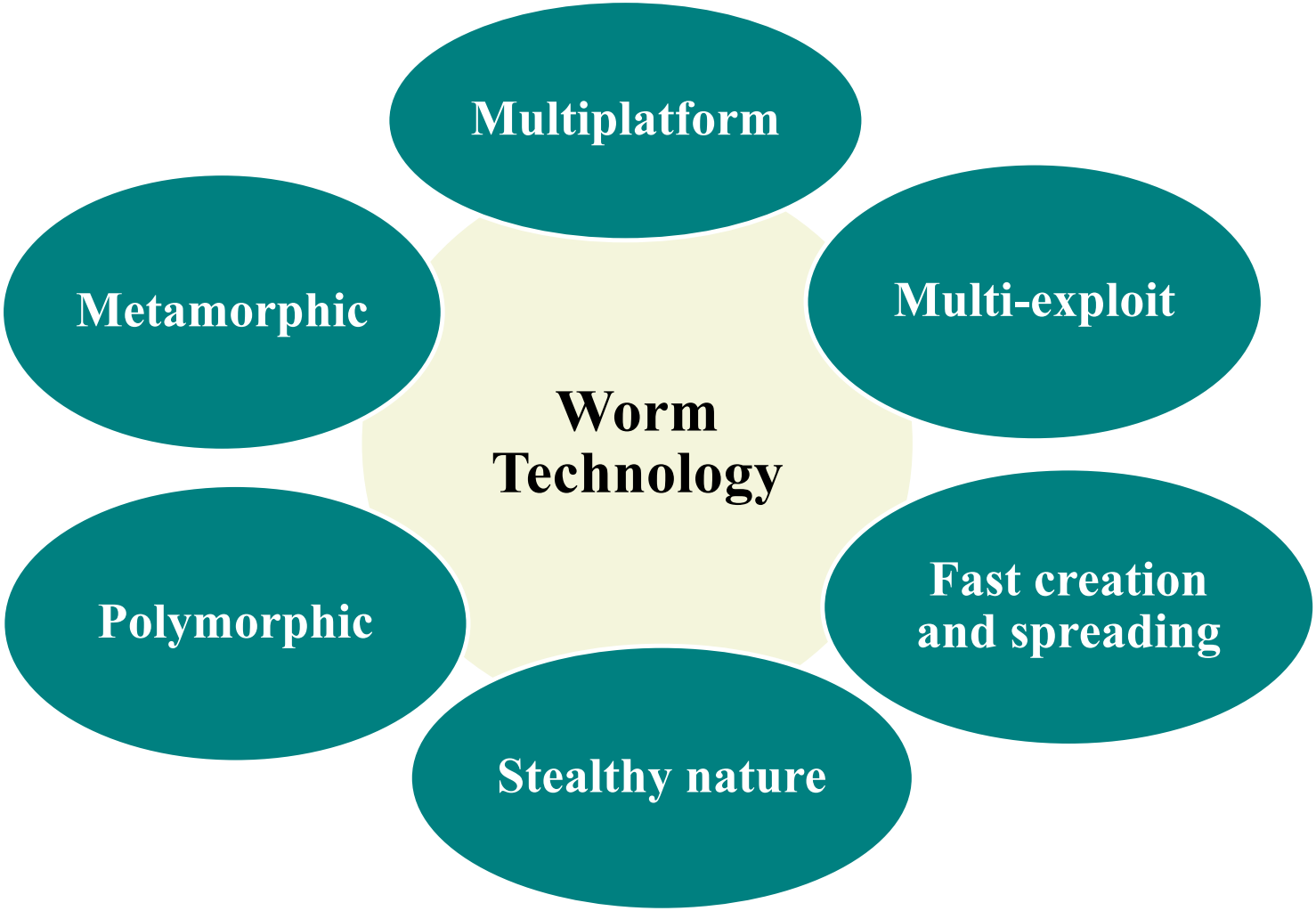
Worms - Propagation

- Typically by exploiting
  - Communication/networking facilities or services, e.g. email, RPC (remote procedure call), web requests.
  - Vulnerabilities or bugs in operating systems, services and applications, e.g. **buffer overflow**, open proxy servers.
  - Mobile code characteristics, e.g. cross-site scripting, email attachment, interactive/dynamic content, etc.
  - Insecured file sharing facilities.

Worms – Activation and Payload

- Activation
  - Human activation, e.g. opening an email attachment, or click on a web link.
  - Human activity based activation, e.g. logging in, rebooting.
  - Scheduled process activation, e.g. updates, backup.
  - Self activation.
- Payload - typically
  - DoS (Denial of Service) attack by consuming bandwidth and other resources
  - Command and control, e.g. botnet
  - Data theft
  - Data integrity attack

Worms - the State-of-the-Art Worm Characteristics



More Malware Types

- **Trapdoor (backdoor)**
  - Secret entry point into a host.
  - Bypass security controls, allowing unauthorised access or compromise of a system.
- **Trojan horse**
  - A self-contained, nonreplicating program appearing to be benign, but with a hidden malicious function.
  - Often used to install a trapdoor or propagate malware.
- **Malicious Mobile Code**
  - Malicious software transmitted from a remote host to a local host and then executed on the local host.
  - Languages used for malicious mobile code include Java, ActiveX, JavaScript, and VBScript.

More Malware Types

- **Zombies (bots)**
  - Use the Internet to gain access or infect a collection of devices (computers, smart phones, etc); these infected devices are called zombies or bots.
  - Users/owners of the infected devices are typically unaware that they are part of the botnet.
  - The bots take instructions from a remote command/control machine operated by the attacker.
  - Zombies are often used to conduct DDoS (distributed denial-of-service) attacks.
- **Blended Attacks**
  - These attacks use multiple infection and/or transmission methods, e.g. web-based malware, drive-by-download, …

Advanced Persistent Threats (APTs)

- Trend to use virus-creation and attack toolkits (called crimeware)
  - Include a variety of propagation mechanisms and payload modules.
  - Create a significant problem for those defending systems against them.

- Change from attackers being individuals (often motivated to demonstrate their technical competence to their peers) to more organized and dangerous attack sources, e.g. APT (Advanced Persistent Threats).

APTs

- Well-resourced, persistent application of a wide variety of intrusion technologies and malware to selected targets (usually business or political).
- Of stealthy nature: quietly, slowly spread to other hosts, gathering information over extended periods of time before data exfiltration and other negative impacts.
- Typically attributed to state-sponsored organizations and criminal enterprises.
- Differ from other types of attack by their careful target selection and stealthy intrusion efforts over extended periods.

APTs

- A variety of techniques used, e.g.
  - Social engineering
  - Spear-phishing emails
  - Drive-by-downloads from selected compromised websites likely to be visited by personnel in the target organization
- Intent includes
  - to infect the target with sophisticated malware with multiple propagation mechanisms and payloads.
  - once they have gained initial access to systems in the target organization a further range of attack tools are used to maintain and extend their access for attacking legitimate services, stealing information, etc.

Defence Measures

- **For individual users,** be cautious, e.g.
  - Do not open any unknown/suspicious file/attachments.
  - Do not click on unknown/suspicious hyperlinks and web browser popup windows.
  - Do not open files with file extensions that are likely to be associated with malware (e.g., .bat, .com, .exe, .pif, .vbs)
  - Do not disable malware security control mechanisms (e.g., antivirus software, content filtering software, personal firewall)
  - Do not use administrator-level accounts for regular host operations.
  - Always use official download sources.
  - Use a legitimate and update anti-virus/anti-malware tool.
  - Always backup your files.

Defence Measures

- **For organisations, I expect you to read this document:**
  - NIST Special Publication 800-83, "Guide to Malware Incident Prevention and Handling for Desktops and Laptops", *https://csrc.nist.gov/publications/detail/sp/800-83/rev-1/final*
- **For software developers**
  - Write bug-free programs, or make it as bug-free as possible.
  - Do rigorous software testing.
- **For software/application distributors**
  - Use detect/remove or detect/alert tools, e.g. to detect any suspicious signs and alerts the customers.
  - Protect the integrity of code, e.g. allow authors to sign the code and for the authors to be listed as trusted.

Part 2 – Overview

- Buffer Overflows
  - What is Buffer Overflow and Why it is Dangerous
  - Program Memory Layout
  - Stack Buffer Overflow Attacks
  - Defence against the Attacks

What is Buffer Overflow

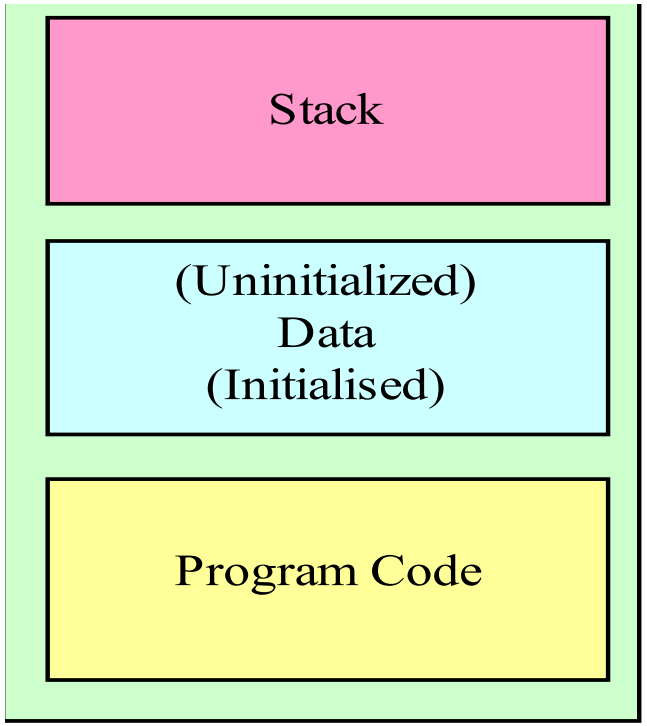
- Buffer is a pre-allocated data storage area inside computer memory.
- Buffer overflow (also called buffer overrun) occurs when the size of the data exceeds the size of the buffer allocated to the data.
- This can happen when a developer
  - uses an unbounded string functions, and
  - forgets to keep track of the size, or makes a mistake with the size.
- Buffer overflow can occur at all levels of code implementation, e.g. improper use of programming languages, libraries, operating systems, and in the application logic, …

What is Buffer Overflow

```
#include <stdio.h>
int main (void)
{
    char str[20];

            printf ("Enter your name: ";
            gets (str);
            printf ("Your name is %s", str);
            return(0)
}
```

Space of 20 characters is reserved in the memory



If the user enters more than 20 chars, then the space of str would overwrite other part of the memory.

Why it is Dangerous

- Consequence:
  - In a normal case, this buffer overflow vulnerability would cause the program to crash.
  - BUT attackers can exploit this vulnerability to gain a complete control of the execution of a program, to execute malicious code, and if the program runs with privileges, attackers may also gain those privileges.

Why it is Dangerous

- Imagine: there is a login procedure that sets the Authenticated flag if the user supplies the correct password (i.e. the authentication is successful); assume the compiler stores the Authenticated variable immediately after str. An attacker may set the Authenticated flag by supplying 21 bytes of data (the 21st byte takes a non-zero value), allowing the attacker to bypass the authentication control.

- Also imagine: if the memory area after str is a function pointer (i.e. address), the attacker could overwrite the area with an address of his own code (malicious code), thus directing the execution flow to his chosen memory address.

```
char str[20];
int authenticated = 0;

void foo()
{
        gets (str);
        …
        bar()
}
```

Why it is Dangerous

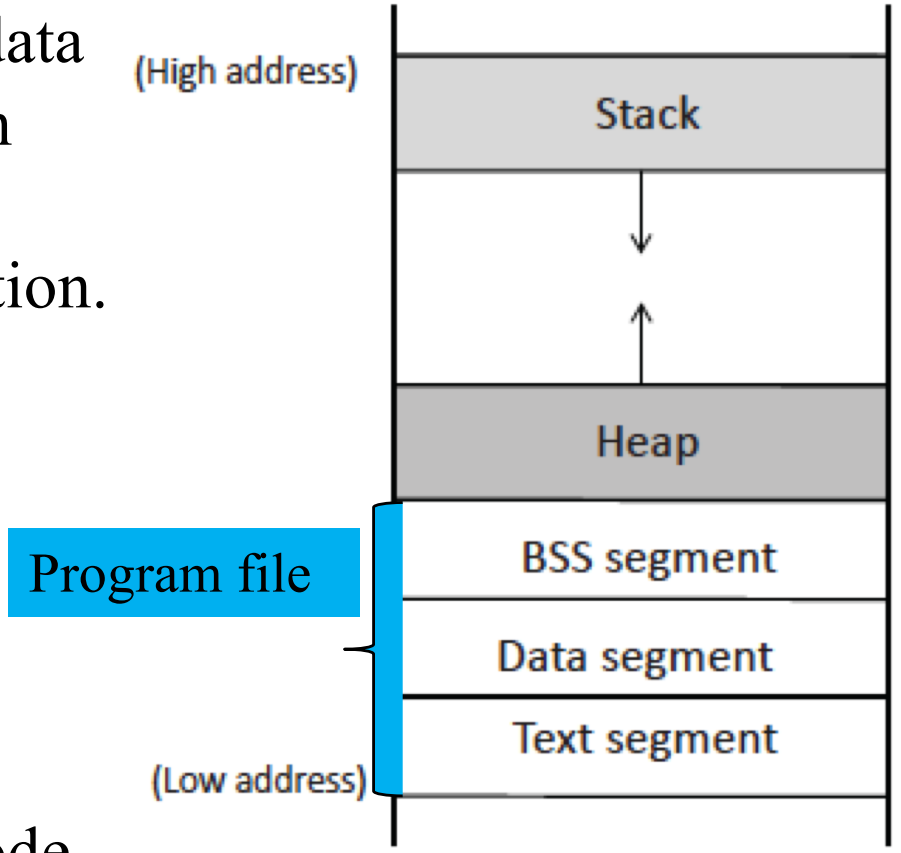
- Buffer overflow is one of the oldest and the most common type of software security weaknesses or vulnerabilities.
- It remains a common source of vulnerabilities and exploits today, especially in embedded systems.
- Some notable examples
  - In 1988, **Morris Worm** infected 10% of the Internet in two days by exploiting a buffer overflow vulnerability in the Unix sendmail, finger, and rsh/rexec, causing tens of millions worth of damage.
  - In 2003, **SQL Slammer** infected Microsoft SQL Server Resolution Service by exploiting a buffer overflow vulnerability in Microsoft's SQL Server and Desktop Engine database products, caused DoS attacks on some Internet hosts, ISPs and ATMs.

Why it is Dangerous

- In 2014, **Heartbleed** exposed data and services of millions users and online services due to a buffer overflow vulnerability in OpenSSL, a popular cryptographic software library.
- In 2016**,** a buffer overflow vulnerability was found in **Adobe Flash Player** for Windows, macOS, Linux and Chrome OS, bypassing security restrictions, execute arbitrary code, and obtain sensitive information.
- In 2019, a buffer overflow vulnerability in **WhatsApp VOIP** stack on smartphones was exploited to infect over 1,400 smartphones with malware ...
- …

Program Memory Layout

- **Stack** for storing local variables, data related to function calls, e.g. return address, arguments, …
- **Heap** for dynamic memory allocation.
- **BSS (Block Start by Symbol) segment** stores uninitialized static/global variables.
- **Data segment** stores static/global variables initialized by the programmer.
- **Text segment** stores executable code of the program.



Program Memory Layout

- A buffer can be a heap or stack.
- Types of buffer overflow attacks
  - **Heap-based attacks**: these are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.
  - **Stack-based attacks (stack overflows):** these are more common, and leverage stack memory that only exists during the execution of a function.

About Stack (based on x86 Processors)

- Important pointers
  - **Instruction Pointer** (register %eip): points to the next instruction it executes.
  - **Stack Pointer** (%esp): points to the last value pushed onto the stack.
    - As values are pushed onto the stack, the stack pointer decreases.
    - Stack is portioned into regions, one per function, called Stack Frames.
  - **(Current) Frame Pointer** (%ebp): points to the (previous) frame pointer of the caller function.

About Stack

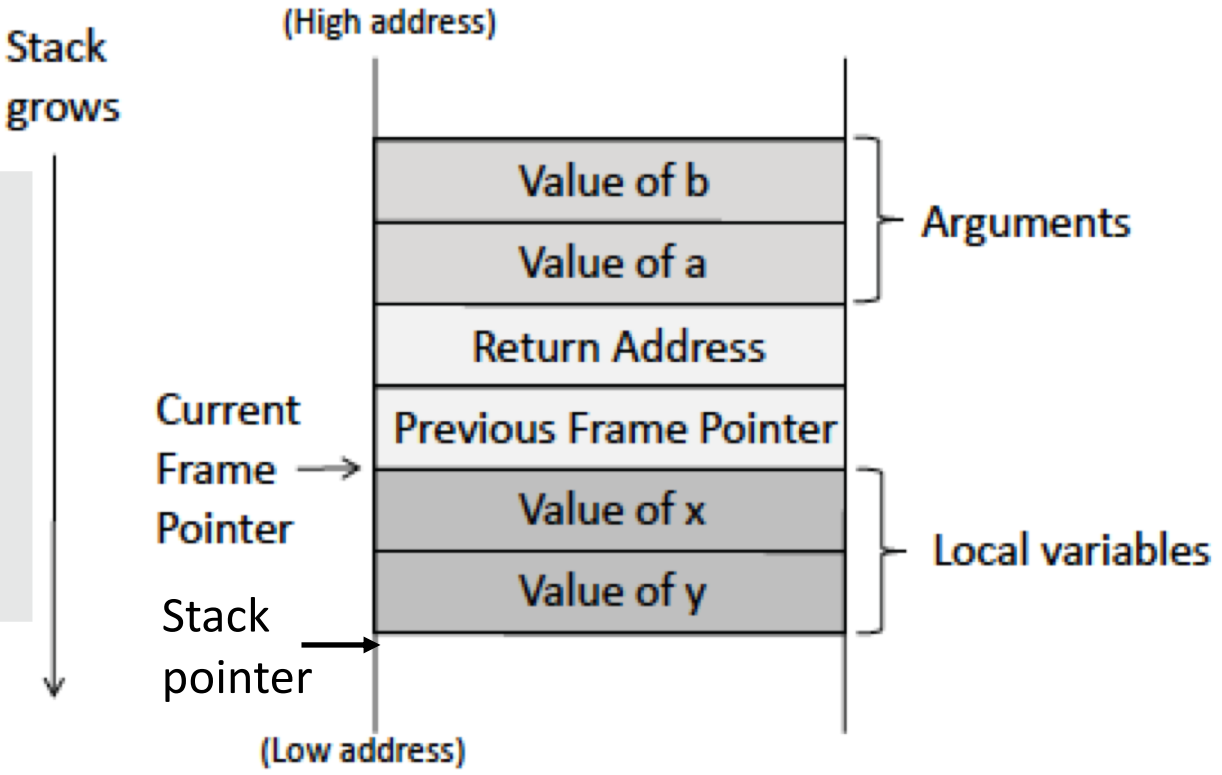
- Calling a function
  - When a function is called, %eip value (Return Address) is pushed onto the stack.
  - When the function is returned, the value pointed by %esp is copied into %eip so execution resumes from this point.
- Arguments of a function
  - The arguments of a function are pushed onto the stack just before the function call. If there is more than one argument, the last argument is pushed on first.
- Frame Pointer (FP): each function has a stack frame and each frame has a FP, a special register pointing to a fixed location in the stack frame on top of the stack, so that the address of each argument and local variable on the stack frame (of the function) can be calculated using FP and an offset.

About Stack

- A stack frame consists of four regions: Arguments, Return Address (i.e. address of the next instruction after Ret), Previous Frame Pointer (PFP, the FP of the caller function), and Local Variables.
- This is the layout of the stack frame for func().

```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

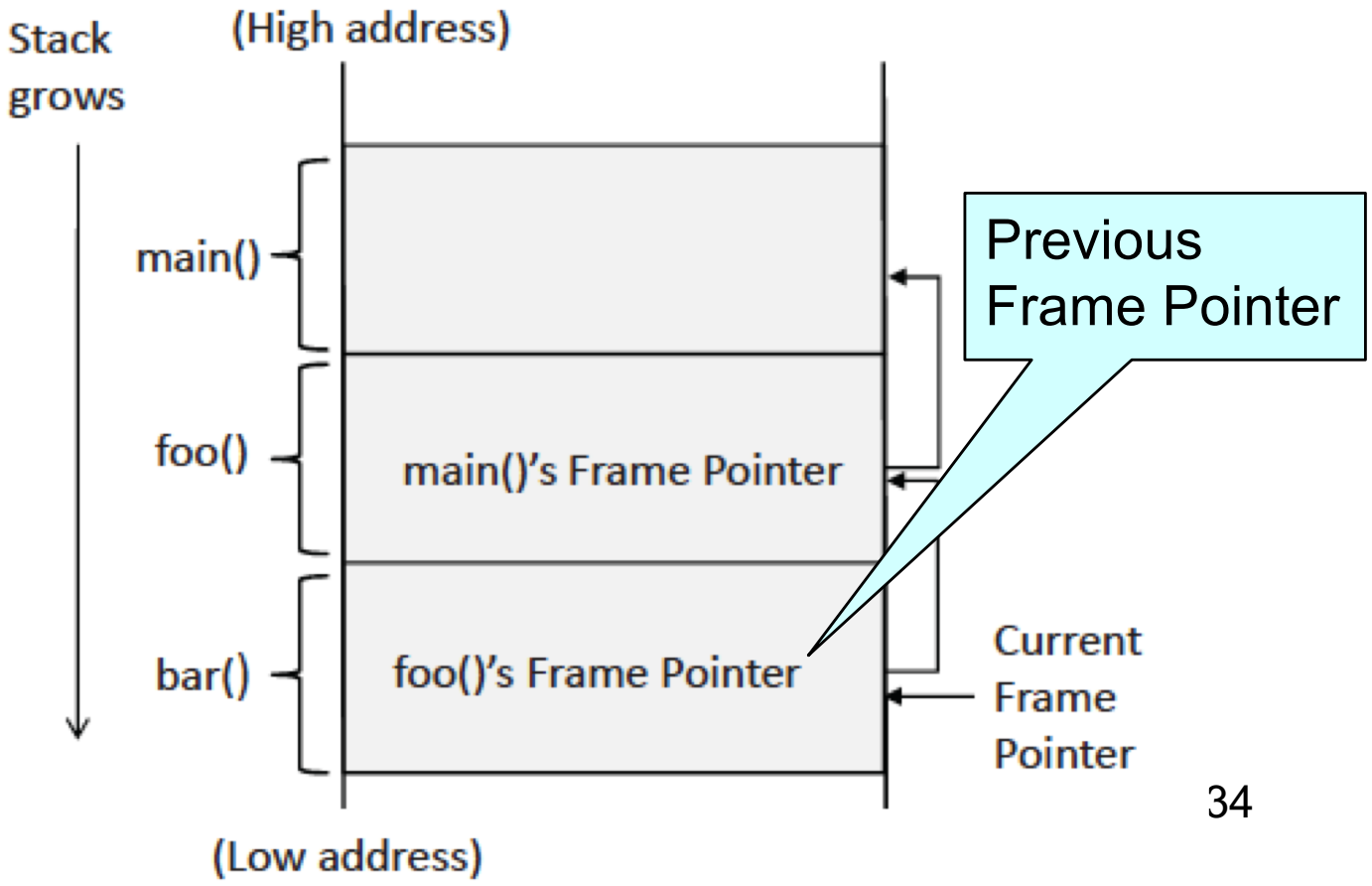


About Stack

- The stack layout for function call chain: for each function call, a stack frame is assigned.
- Stack grows and shrinks as functions are called and returned.

```
main ()
{
   foo()
   …
}

void foo()
{
    bar()
    …
}
```



About Stack

- Before entering bar(), foo()'s FP value is stored in the "previous frame pointer" field of bar()'s stack frame. When bar() returns, the value in this field will be used to set the FP register, making it point to foo()'s stack frame again.

- Similarly, main()'s frame pointer value is stored in the "previous frame pointer" field of foo()'s stack frame.

About Stack - Summary

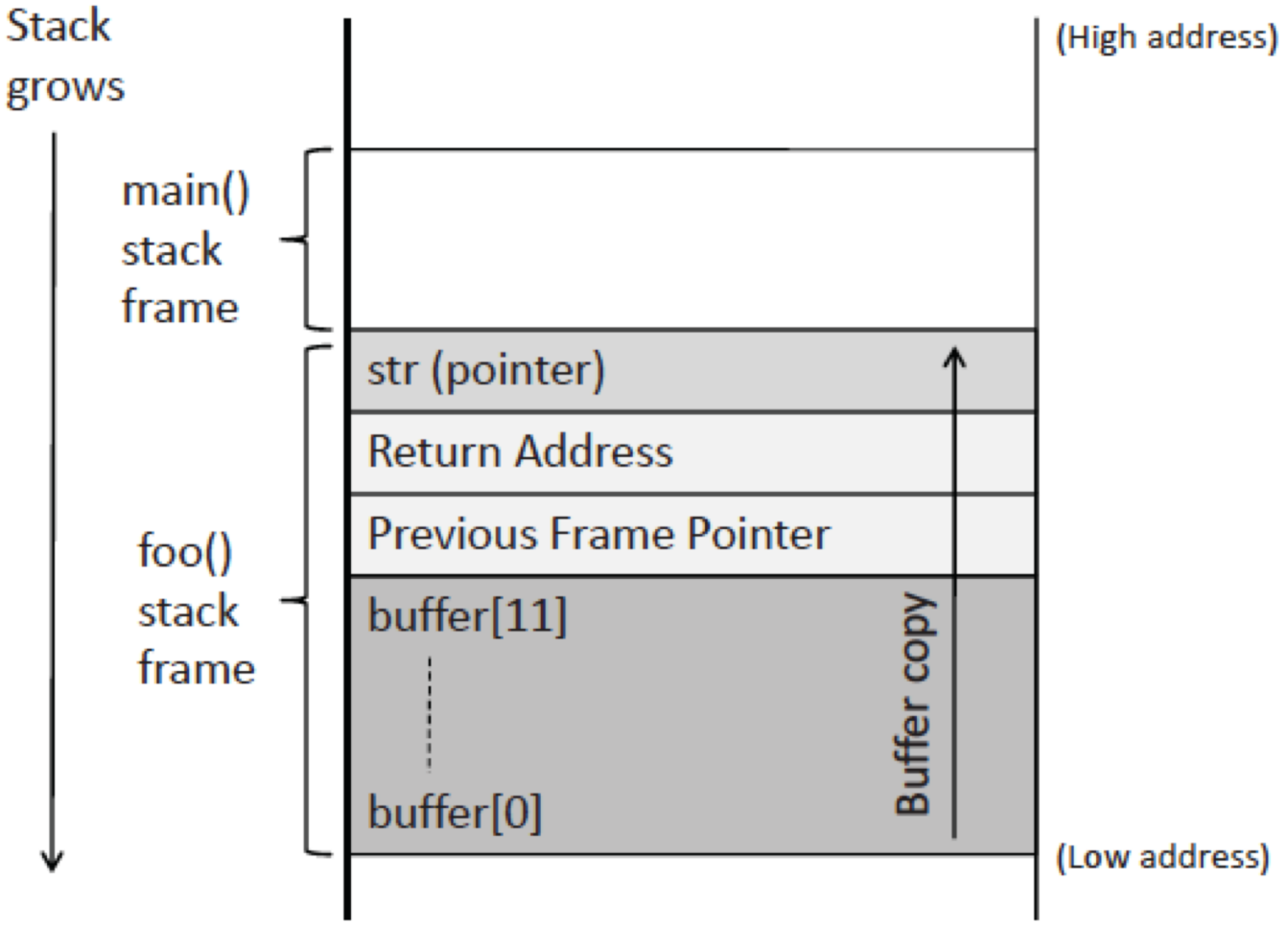
- Arguments are pushed onto the stack before a function call.
- Stack Pointer (%esp) points to the last item in the stack.
- Instruction Pointer (%eip) points to the next instruction to execute.
- Call <addr> pushes the current value of %eip and sets %eip to <addr>.
- Ret pops the Return Address from the stack into %eip and execution continues ...
- Frame Pointer (%ebp) points to the frame pointer of the caller function.

Stack Buffer Overflow Attacks

How this may be exploited and think about the consequences.

```
void foo(char *str)
{
    char buffer[12];
    strcpy(buffer, str);
}

int main()
{
    char *str="this is a string
that is longer than 12";
    foo(str);
    return 1;
}
```



Stack Buffer Overflow Attacks

- strcpy() is null terminated; it does not do input size checking.
  - strcpy(buf, str) copies memory contents into buffer starting from *str until "\0", or the end of string, is encountered.

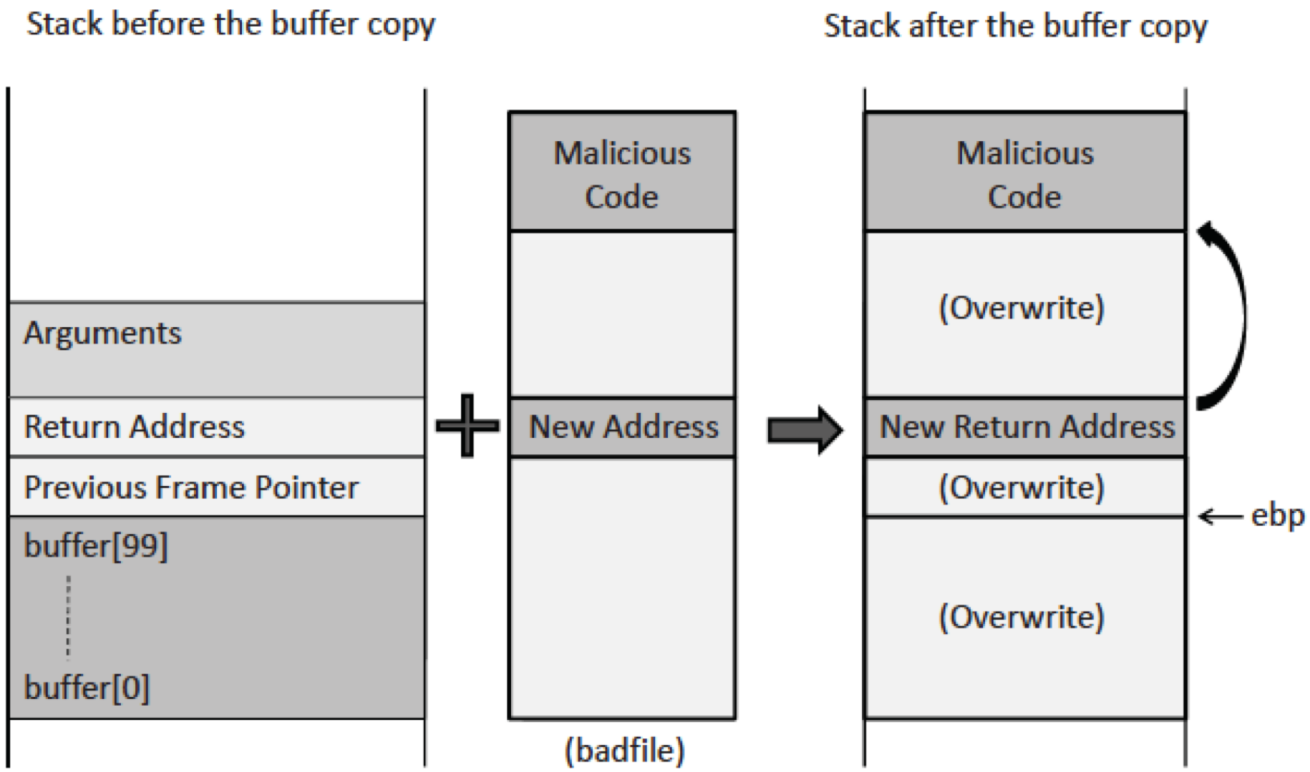
- If str is longer than the buffer size,
  - Program could crash.
  - Attacker can change the program behaviour, executing different code – the situation becomes more serious if the 'different code' is provided by attackers.

Stack Buffer Overflow Attacks

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

int foo(char *str)
{
    char buffer[100];
    strcpy(buffer, str);
   return 1;
}

int main(int argc, char **argv)
{
   char str[400];
   FILE *badfile;
   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 300, badfile);
   foo(str);
   …
}
```



In the proper place of **badfile**, it contains '**New Return Address**' pointing to the **Malicious Code** which is placed at the end of badfile.

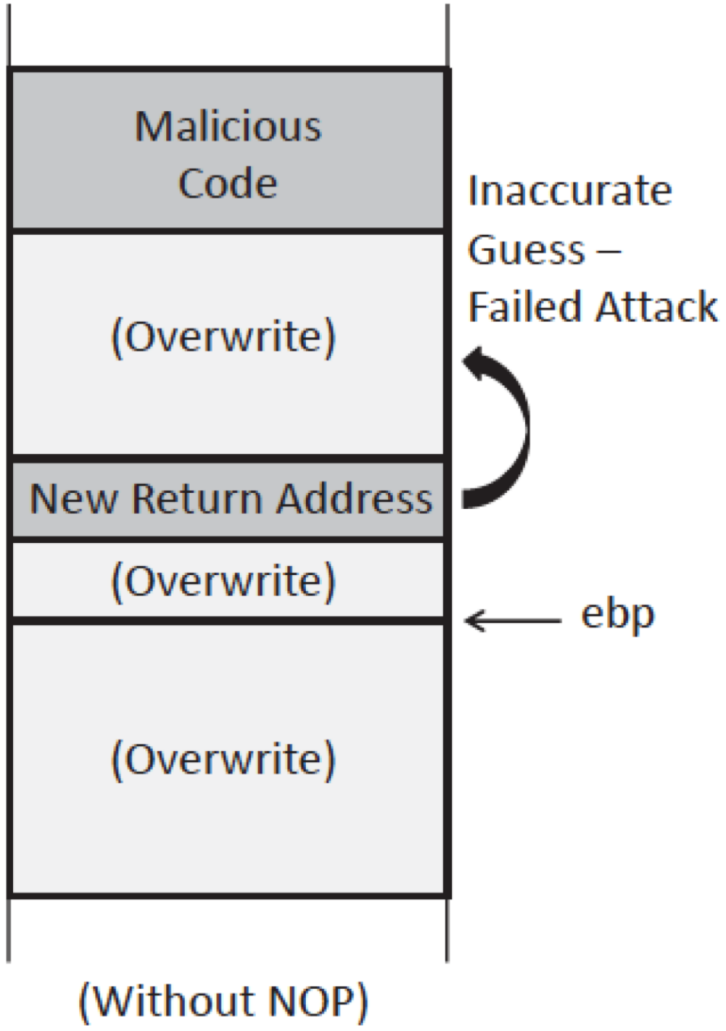
Stack Buffer Overflow Attacks

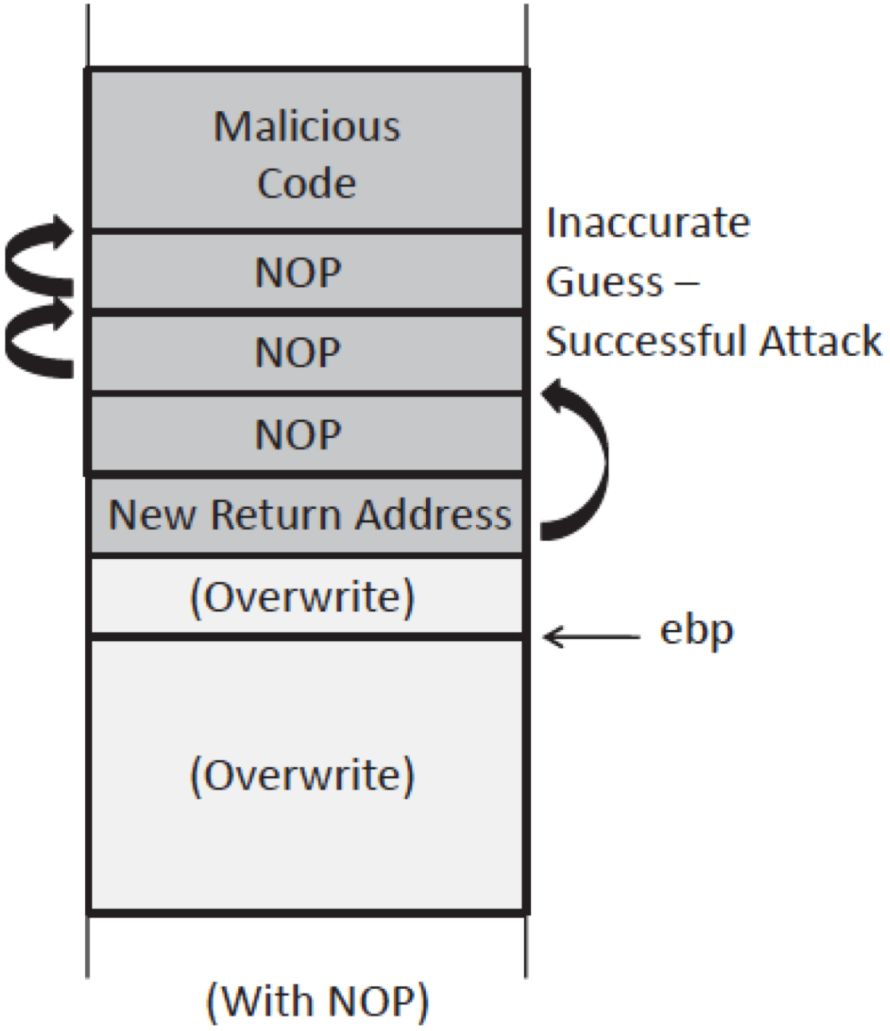
- To perform a successful stack buffer overflow attack, the Return Address in the overflowed stack frame need to point to the starting address (entry point) of the malicious code.
- To achieve this, the attacker need to guess the entry point of the malicious code.

Stack Buffer Overflow Attacks

- **Assumptions**: (i) memory unit is 32-bits, (ii) buffer Starting Address is X, buffer size is H **(buffer address range is [X, X+H]).**
- Need to know where in the input string should be used to hold Return Address (RA) and the Offset between RA and the malicious code.
- Approach 1: Guess X and calculate RA (if target program is known)
  - Turn off OS countermeasures such as *address randomisation facility.*
  - Guess X, e.g. by printing out the address of some local variable in the function or using debugging facility.
  - Analyse target program to work out location of RA and make it (New RA) point to entry point of malicious code.
  - Use No-Op (NOP) instructions to create multiple entry points; NOP instruction advances the program counter to the next location.

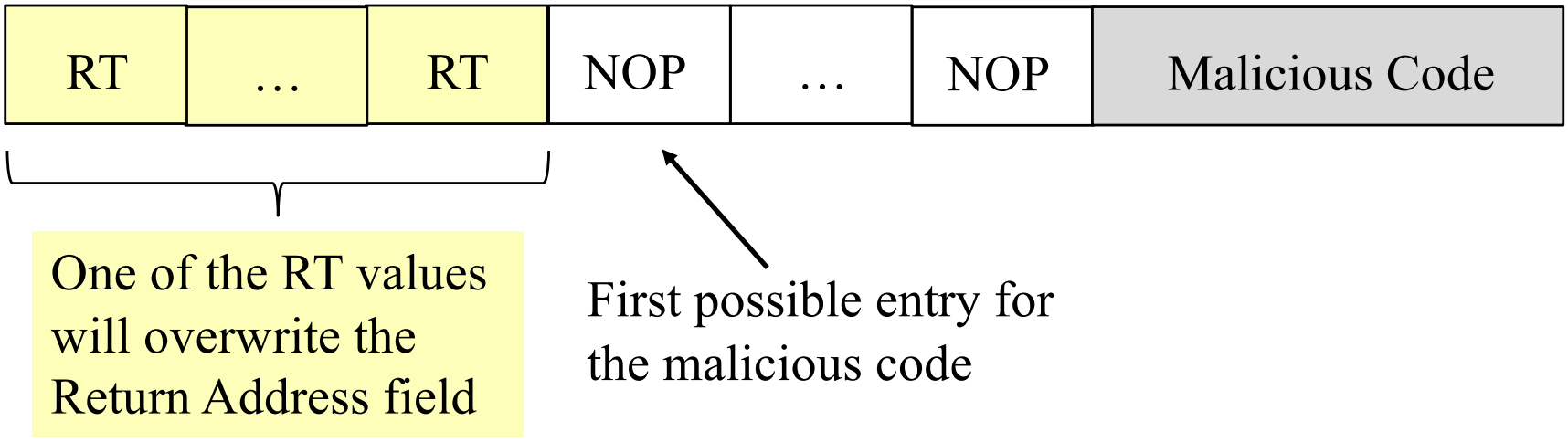
Stack Buffer Overflow Attacks





Stack Buffer Overflow Attacks

- Approach 2: trying the spraying technique, i.e. spray the buffer with return address (if target program is not known).
- The figure below shows the structure of a malware.



Defence against Buffer Overflows

- **Programming language selection**
  - Choose a programming language that are not or less vulnerable to buffer overflow attacks.
  - But for certain applications, there are other considerations such as resource constraints, and execution speed.
- **Safe coding techniques**
  - Clearly define the sizes of arrays/variables.
  - Ensure the length of the value stored in a variable is not greater than the variable can hold.
  - Avoid the use of unbounded reads.
  - Use safer libraries/variants.
  - Validate input to ensure that the amount of data read fit within the allocated buffer space.

Defence against Buffer Overflows

- **Stack protection mechanisms**
  - Stackguard: A canary value (typically a random value) known to the program is inserted before the Return Address on the stack on function entry, and compared against on function exit.
  - Stackshield and Return Address Defender (RAD): A copy of the return address is made on function entry and compared against on function exit.
- **Data execution prevention:** Mark stack (and heap) nonexecutable.
- **Address space randomization:** Randomize the address where the stack is located for each process, the memory address where a particular library will be imported, etc.

Exercise Question – E11.1

(a) Investigate different forms of malware and types of attacker tools.
(b) To protect a business (a company or an organisation) against malware attacks, what are the main elements of prevention? Justify your answer.

Exercise Question – E11.2

(a) Name some library functions in a programming language of your choice, which are vulnerable to buffer overflow attacks.
(b) With regard to what you identify in (a), give some countermeasures to mitigate the risks.

Conclusions

- There are various malicious programs with various purposes. They are spread via social engineering, resource sharing applications/services, software vulnerabilities, and system and network backdoors, etc.
- Buffer overflows are among the most commonly seen software vulnerabilities.
- They are exploited to overwrite memory values.
- The written values can allow an attacker to execute their own code mounting malicious attacks.
- Buffer overflows are caused by programming mistakes and/or using unsafe library functions.