6 System Security

Syllabus

At the end of this unit, you should be able to understand and comprehend the following syllabus topics :

- Software Vulnerabilities
  - Buffer Overflow
    - Format string
  - SQL injection
- Malware
  - Viruses
  - Worms

6.1 Software Vulnerabilities

Q. List various Software Vulnerabilities. How vulnerabilities are exploited to launch an attack? **MU - May 19, 5 Marks**

Today it is hard to imagine any software without any vulnerabilities. Some vulnerabilities are found during software testing (for example penetration testing, static code analysis, dynamic code analysis, etc.) and others are found in the field (when the software is put to mass use). Software vulnerabilities that are found after the software has been released are mitigated (fixed) by giving out software patches or updates. You would have seen your browser, operating system and apps updating multiple times in its course of lifetime. These updates include both feature updates (new functionalities in the software) and also security updates (fixing new vulnerabilities found in the software).

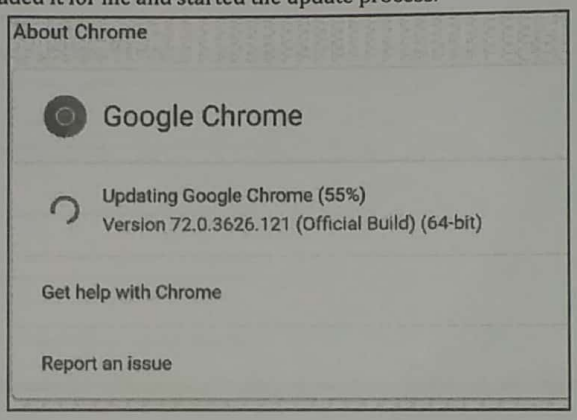
Software developing companies have security vulnerability update programs and teams that regularly investigate all reported security vulnerabilities form the field. As you learnt in the previous chapter, these companies also run bug bounty programs that help them find security vulnerabilities in their products and mitigate them before these vulnerabilities could be mass exploited.

Let's see an example, Depending on the type of vulnerability, it can be exploited either manually or using scripts or tools. Attacker has deep understanding of how the vulnerability works and she carries out the required steps in the right sequence to exploit the vulnerability. How a particular vulnerability is exploited really depends on the type of vulnerability. There is no fix mechanism. It depends on the complexity of the vulnerability and also on the skills of the attacker.

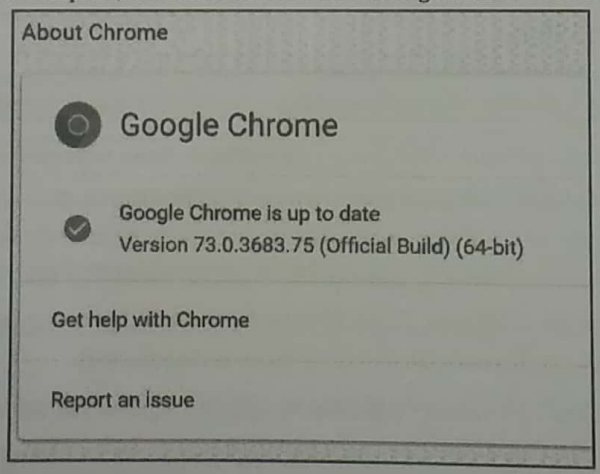
[An interesting note – I was writing this section on 17-Mar-2019. I opened the Google Chrome browser on my laptop to give you an example of security updates in the Google Chrome browser and you know what? When I opened the browser, I found that an update was waiting to be installed. It gave me a smile.

It would make a perfect example for you! It showed how security vulnerabilities are updated regularly in the products to avoid mass exploits. I stopped writing for a while and updated the browser that contained several security fixes. Writing can wait, security fixes cannot. I also grabbed a few snapshots for you meanwhile.]

As soon as I went to the version section of the Google Chrome browser, the browser automatically found that a new update was waiting and downloaded it for me and started the update process.

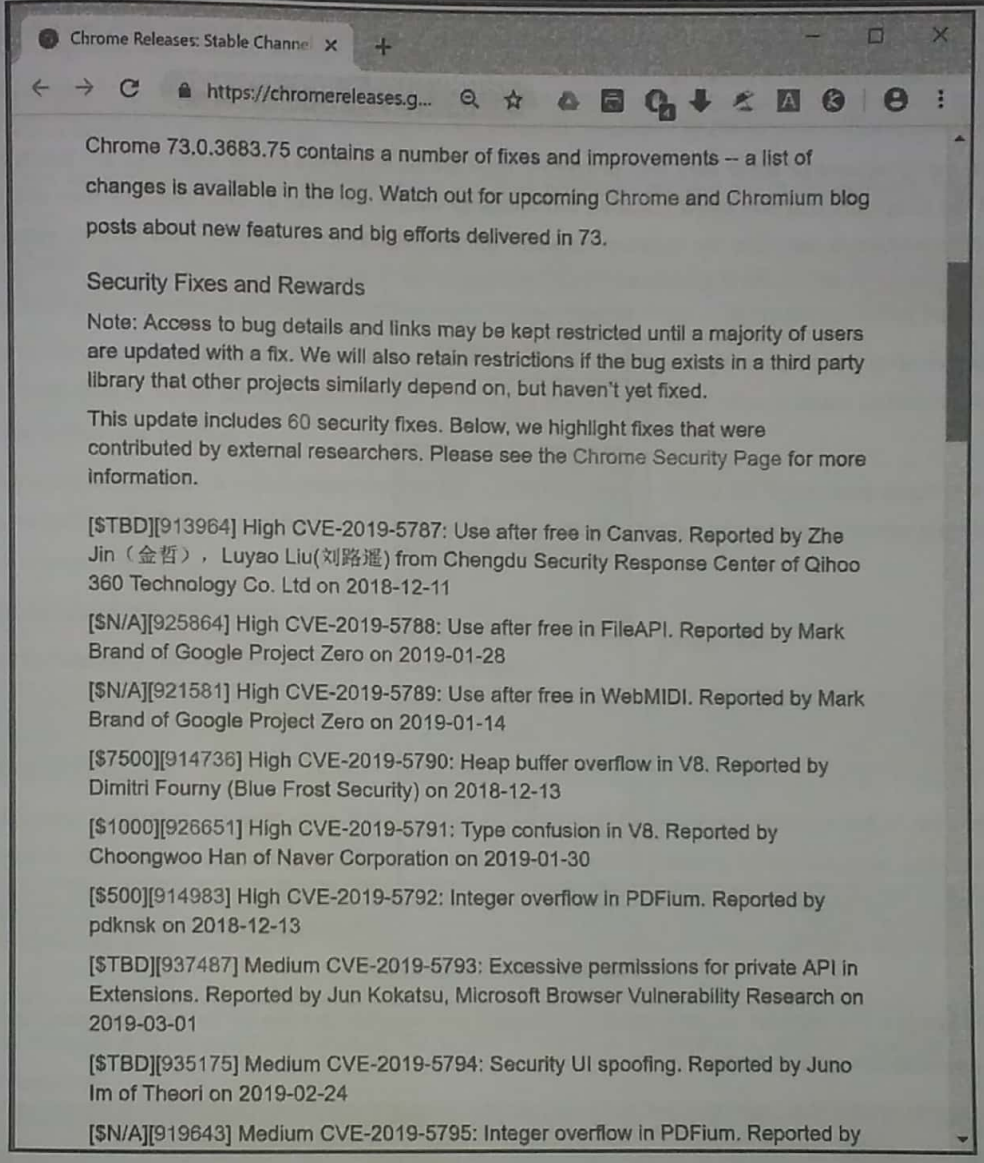


- Once the update process was complete, it showed me that I am running the latest version of Google Chrome.



Now, I wanted to find out what security updates were installed as part of the version 73.0.3683.75. The Chrome release website had the complete listing of the security vulnerabilities that were fixed in the version 73.0.3683.75.

https://chromereleases.googleblog.com/2019/03/stable-channel-update-for-desktop_12.html



- Do you see that 60 security fixes were done in this version? Huge, isn't it?
- Do you see the $ values against some of the security vulnerabilities? Those are the bug bounty program rewards that would be given to the security researchers who reported the vulnerabilities to the Google Chrome security team.
- By the way, do you also see a vulnerability by name "buffer overflow"? Smile, you will read about it in this chapter.

Many software vulnerabilities are well-known, and the applications are frequently examined to check if those vulnerabilities exist. You will read about a few such vulnerabilities in this section. Gear up!

6.2 Buffer Overflow

Q. Write in brief about - Buffer overflow attack. MU - Dec. 15, 5 Marks

Q. Write short notes on Buffer overflow. MU - May 19, 4 Marks

Buffer overflow is also called buffer overrun or buffer overwrite. But before you proceed with learning what a buffer overflow is let's understand what a buffer is.

Definition : *Buffer is a section of computer memory for temporarily storing information.*

When you run programs or enter data into programs, they are stored in the buffer until they are finally saved on the hard disk. All the programs that you run are loaded into memory before they can be executed. Any data that the program requires for its operation is stored in the memory as well. This memory space is called virtual address space. The virtual address space is divided into various segments and each segment is designated a specific purpose.

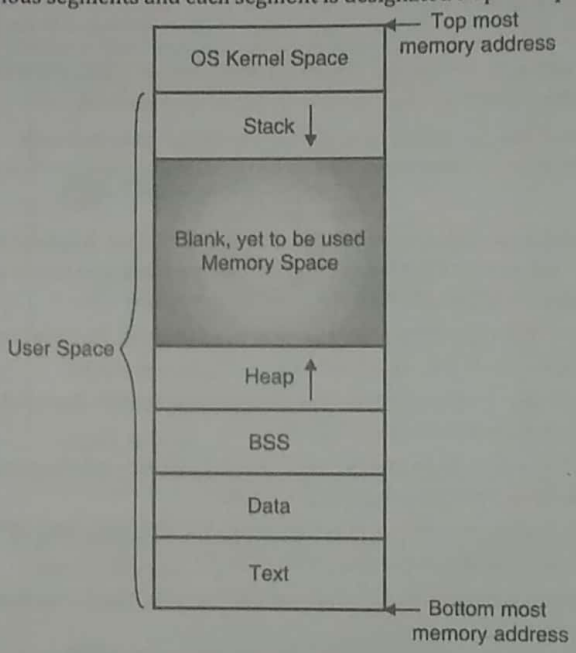


Fig. 6.2.1 : Memory Layout

- **OS Kernel Space :** The topmost memory space is occupied and reserved for the OS Kernel. User programs or data do not have access to this memory area.
- **Stack :** Stack is where the user executable programs are loaded before execution. The Stack grows downwards (towards lower memory addresses) to occupy the unused memory space. The stack pointer references to the top of the stack (the last occupied stack address) all the time.
- **Heap :** Heap is where the values for various dynamic variables, as required by the executing programs, are stored. Heap space is dynamically allotted as per the program needs and the exact heap space is not pre-determined. Heap grows upwards (towards higher memory addresses) to occupy the unused memory space.
- **BSS :** The BSS (Block Started by Symbol) segment, also known as uninitialized data, contains all the global variables and static variables that are initialized to zero or do not have explicit initialization in the program. For example, if you define 'static int a', the value of *a* would be set to 0 and would be placed in the BSS.
- **Data :** The Data segment contains any global or static variables which have a pre-defined value. For example, if you define 'int a = 10', the value of *a* would be set to *10* and would be placed in the Data segment.

- **Text :** Text segment, also known as the Code segment, contains executable instructions of the loaded program. It is read-only to avoid any changes during runtime. Any attempt to write in this segment will cause the OS kernel to kill the process with a segmentation fault error.

Now that you understand the virtual address space and how programs are laid out in the memory, let's define what a buffer overflow is.

Definition : *Buffer overflow is a condition in which the input information crosses the boundary limits or capacity of any segment in the virtual address space thus overwriting information in other segments.*

Adversaries (attackers) exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system. So, for example, an attacker can fill the stack or the heap area so much so that it exhausts all free memory on the system and crash it.

A real-life example could be a water glass that has say a limited capacity of 250 ml. if you keep pouring water in it, the water begins to spill over after its capacity is full. The spilled water could dirty the surface on which the glass is kept or may even damage the surface.

Let's take an example to understand it better.

Assume the following code snippet written in C language.

```
char A[8];
int B=10;
strcpy(A,"EVILINPUT");
```

A is a character array that can hold up to 8 characters. It is followed by an integer *B* that is initialized with the value of 10. Assume that both are adjacent (side by side) in the memory. So, a typical memory layout would be as following.

A								B
								10

Now, the variable *A* is assigned the value of "EVILINPUT", which is 9 characters, without checking the length of the input. What happens?

A								B
E	V	I	L	I	N	P	U	T

This assignment overwrites the value of *B*. The variable *A* should have been restricted to the limit or the boundary that is assigned to it. But that does not happen (especially in the low-level languages like C and C++). The attacker can take advantage of such erroneous coding practices and can potentially exploit the system.

As per National Vulnerability Database (NVD), following is the number of reported vulnerabilities against buffer overflow.

https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=buffer+overflow&queryType=phrase&search_type=all

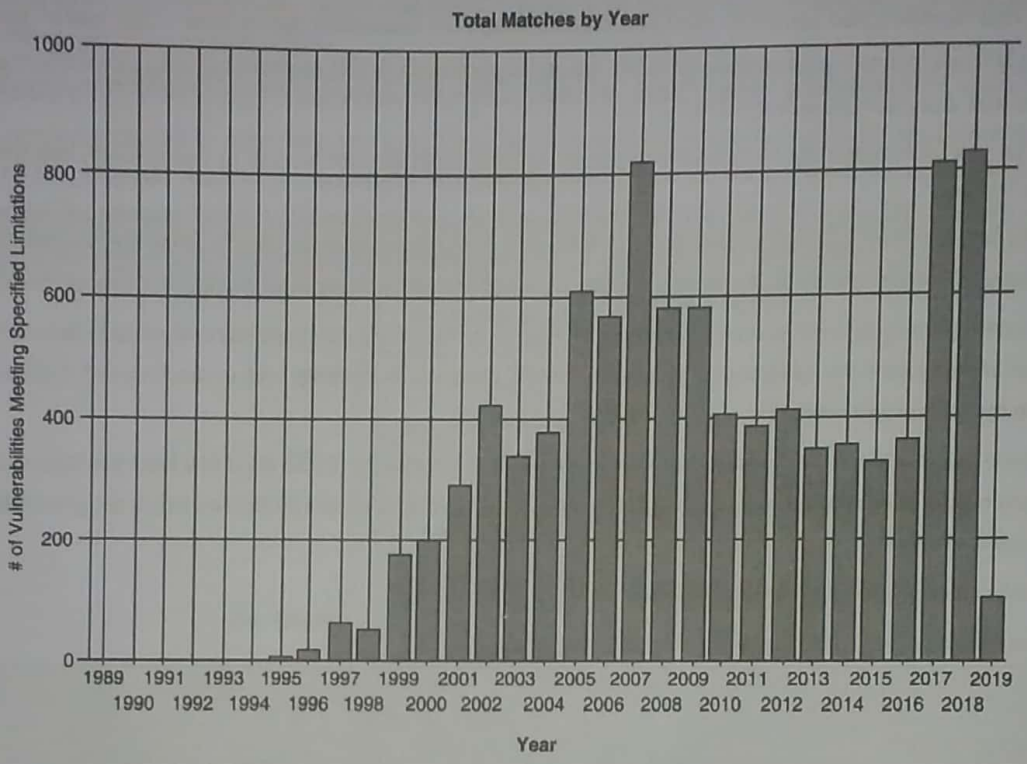


Fig. 6.2.2 : The number of reported vulnerabilities against buffer overflow

Note : Don't get confused by the year 2019. I am taking this snapshot in Mar 2019. The rest of the year is still to pass-by. I am sure that there would be more buffer overflow vulnerabilities discovered in the later months of the year.

Buffer overflow is one of the earliest types of vulnerabilities (first reported in 1988 as part of the Morris Worm) found in software and is still found very frequently even in the latest applications such as Google Chrome as you learnt earlier.

6.2.1 Causes of Buffer Overflow and Suggested Protection Mechanism

There can be several causes that can lead to buffer overflows. Let's understand them.

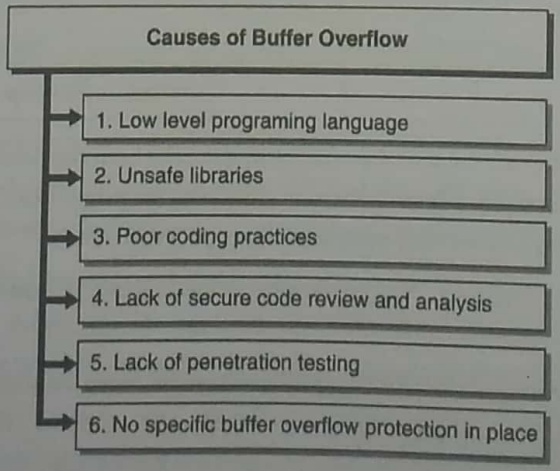


Fig. 6.2.3 : Causes of Buffer Overflow

6.2.1(A) Low-level Programming Languages

There is an old saying that with great power comes great responsibility. Low-level languages such as C and C++ give a lot of power to the developers to handle the executional aspects of their programs such as memory allocation and garbage collection. Also, these languages were developed at times when the programming community was just using assembly language, and these were considered to be very close to assembly language in terms of performance. Also, those times software attacks weren't considered such prevalent (widespread) enough.

Unlike the modern programming languages, C and C++ do not check for overflows themselves. If your program does not check for the overflow conditions, the CPU will raise no errors or warnings and would allow overflow to happen. Also, in C and C++, arrays are compile-time entities only. At execution time, there are only pointers. Hence, there is no method to check any out of bound access at the program runtime.

Hence, until and unless, you are writing system level applications, it is recommended to choose modern programming languages.

Language/Environment	Direct Memory Access	Safety from buffer overflow
Java	Not allowed	Safe
.NET	Not allowed	Safe
Perl	Not allowed	Safe
Python	Not allowed	Safe
Ruby	Not allowed	Safe
C	Allowed	Unsafe
C++	Allowed	Unsafe
Assembly	Allowed	Unsafe

These languages do not allow buffer overflows. If attempts are made to exceed the defined boundaries, these raise exceptions (runtime errors) and terminate the program. So, an attacker might not be able to proceed further and carry out attacks.

6.2.1(B) Unsafe Libraries

In programming terms, a library is a collection of pre-written programs, routines, system calls, functions, classes, metadata, etc. that you can use when you are writing your programs. You don't have to write code yourself to carryout primitive (basic) tasks such as taking an input from the user or printing output on screen.

As you understand, these libraries are compiled from various other programmers like yourself. These libraries might not have sufficient checks for buffer overflow protection. Use of such libraries can make your program vulnerable to buffer overflow as well. Wherever possible, you should look for alternative libraries that have built-in buffer overflow protection.

Following is the sample list of C functions not having buffer overflow protection and their respective safer alternatives. You should be extremely cautious when using the unsafe functions and prefer using safer alternatives.

Unsafe function	Safer alternative
gets()	fgets()
scanf()	scanf_s()
strcpy()	strlcpy()
strcat()	strlcat()
sprintf()	snprintf()
vsprintf()	vsnprintf()

Your applications written in C should use the secure methods provided by the Safe C Runtime library, also known as Safe CRT. Also, C++ applications should use the Standard Template Library classes, often called STL, which automatically detect overflow conditions and throw an exception when such a condition occurs.

6.2.1(C) Poor Coding Practices

If you are programming in a language like C, that gives you direct interaction with the system memory, you need to be more careful. Memory allocation, memory deallocation, pointer address, return address, buffer size, buffer operations, etc. are something that you need to do yourself without getting much assistance from the underlying programming language or the system. Let's take an example.

Assume the following code snippet written in C language.

```
char A[8];
int B=10;
strcpy(A,"EVILINPUT");
```

What is a better alternative?

```
char A[8];
int B=10;
strlcpy(A,"EVILINPUT", sizeof(A));
```

You replaced *strcpy()* with *strlcpy()* that allows you to control precisely that irrespective of what size of input is given to the variable *A*, only the input up to the capacity of the variable *A* would be considered. This is much better when compared to the previous code when it comes to protecting from buffer overflow vulnerability. You must do all sorts of input validation:

- Length checks
- Boundary checks
- Data type checks

As a programmer, you need to exercise discipline when writing code specially with languages that give you a lot of power and system level direct interaction capabilities.

6.2.1(D) Lack of Secure Code Review and Analysis

You review your responses to examination questions and rectify any oversight issues or mistakes, isn't it? Similarly, as an industrial practice, code is subject to review and analysis. Your peer (colleague) can review what you wrote and can

point out any issues that could be potentially dangerous and should be fixed before putting code into practice. Also, there are a lot of tools that can automate the process of testing your code and identify potential vulnerabilities.

Some of them are,

Software	Language(s)
.NET Security Guard	.NET, C#, VB.net
Agnitio	ASP, ASP.NET, C#, Java, JavaScript, Perl, PHP, Python, Ruby, VB.NET, XML
Brakeman	Ruby, Rails
Checkmarx Static Code Analysis	Multiple
CodeSec	C, C++, C#, Java, JavaScript, PHP, Kotlin, Lua, Scala, TypeScript, Android
CodeSonar	C, C++, Java
Coverity	Multiple
DevBug	PHP
Find Security Bugs	Java, Scala, Groovy
FindBugs	Java
FlawFinder	C, C++
Fortify	Multiple
Google CodeSearch Diggity	Multiple
Klocwork	C, C++, C#, Java
Microsoft FxCop	.NET
Microsoft PREFast	C, C++
ParaSoft	C, C++, Java, .NET
phpcs-security-audit	PHP
PMD	Multiple
Polyspace Static Analysis	C, C++, Ada
Puma Scan	.NET, C#
Puma Scan Professional	.NET, C#
PVS-Studio	C, C++, C#
RIPS	PHP
RIPS NextGen	PHP
SonarCloud	Multiple
Splint	C
Veracode	Multiple
VisualCodeGrepper	C/C++, C#, VB, PHP, Java, PL/SQL

These tools help you to find potential vulnerabilities in your code quickly without you having to go through your code line by line. Usually these tools are used in combination with the manual code review. So, if you are writing some code and putting it directly into practice, it is not recommended. Ensure that it has been reviewed manually as well as scanned by a tool of your choice before you actually put it to use for everyone else. You should cover the following in your code review and analysis.

- Arrays (For example, int a[10])
- Format strings (For example, printf(), fprintf(), %x, %s, %n, %d, %u, %c, %f)
- Functions that are known to be vulnerable to buffer overflow (For example, gets())

6.2.1(E) Lack of Penetration Testing

Recall from the last chapter that the goal of penetration testing is to identify and fix the vulnerabilities before any threat can exploit those vulnerabilities. It involves thinking like an attacker and conducting tests to find out vulnerabilities in the system. Quite a few programs are never penetration tested. Consequently, the attacker might discover such vulnerabilities and can potentially exploit the system. So, before you put your code into practice, it is important to carry out tests to discover any buffer overflow vulnerabilities and fix them. There is a wide variety of tools and methods that can conduct penetration testing to discover buffer overflow vulnerability.

6.2.1(F) No Specific Buffer Overflow Protection in Place

As you know, buffer overflow is quite an old and known vulnerability. There have been several protection mechanisms developed since then to protect the system and applications from exploits. It is highly recommended to have these protection mechanisms in place. Let's learn about them.

6.2.2 Specific Protection Mechanisms for Buffer Overflow

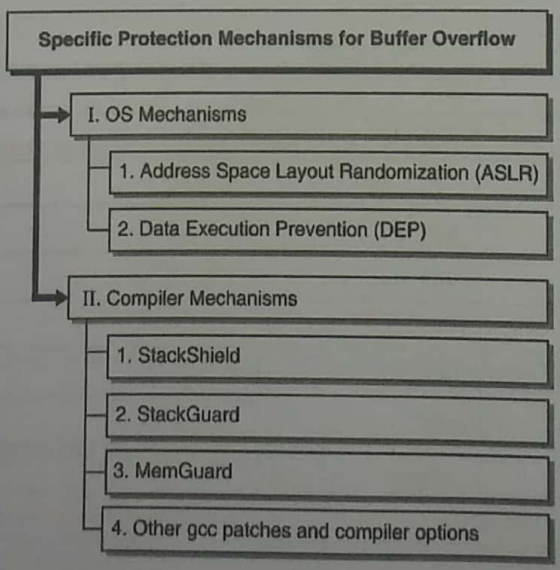


Fig. 6.2.4 : Specific Protection Mechanisms for Buffer Overflow

6.2.2(A) Operating System Based Buffer Overflow Protection Mechanisms

There are certain protection mechanisms that can be applied at the operating system level to avoid buffer overflow attacks. These mechanisms help to secure memory access and provide protection. These mechanisms are independent from the program that you are executing and are applied for all programs globally. Let's take a look at them.

1. Address Space Layout Randomization (ASLR)

To exploit the buffer overflow vulnerability, the attacker must guess the memory address at which the program and its parts are located.

> ***Definition :*** *Address Space Layout Randomization (ASLR) is a memory protection technique that tries to prevent an attacker from guessing the memory addresses by randomizing the memory address layout of various program components each time the program runs.*

The program is loaded at a different base address in memory each time it is executed. Also, the addresses for memory segments like the heap and the stack are randomized. This makes it harder for attackers to guess the correct address and carry out exploits on buffer overflow.

For example, carefully look at the following snapshot. The program WinRAR is loaded at different base address on each invocation. In the first instance, the base address starts with 00DE whereas in the second instance, the base address starts with 0137. Such a randomization makes it hard for the attacker to guess the memory locations of programs and the related objects.

First invocation

00DE0000	00001000	WinRAR		PE header
00DE1000	0010E000	WinRAR	.text	code
00EEF000	0001A000	WinRAR	.rdata	imports
00F09000	000A1000	WinRAR	.data	data
00FAA000	00001000	WinRAR	.gfids	
00FAB000	00001000	WinRAR	.tls	
00FAC000	00037000	WinRAR	.rsrc	resources
00FE3000	0000D000	WinRAR	.reloc	relocations

Second invocation

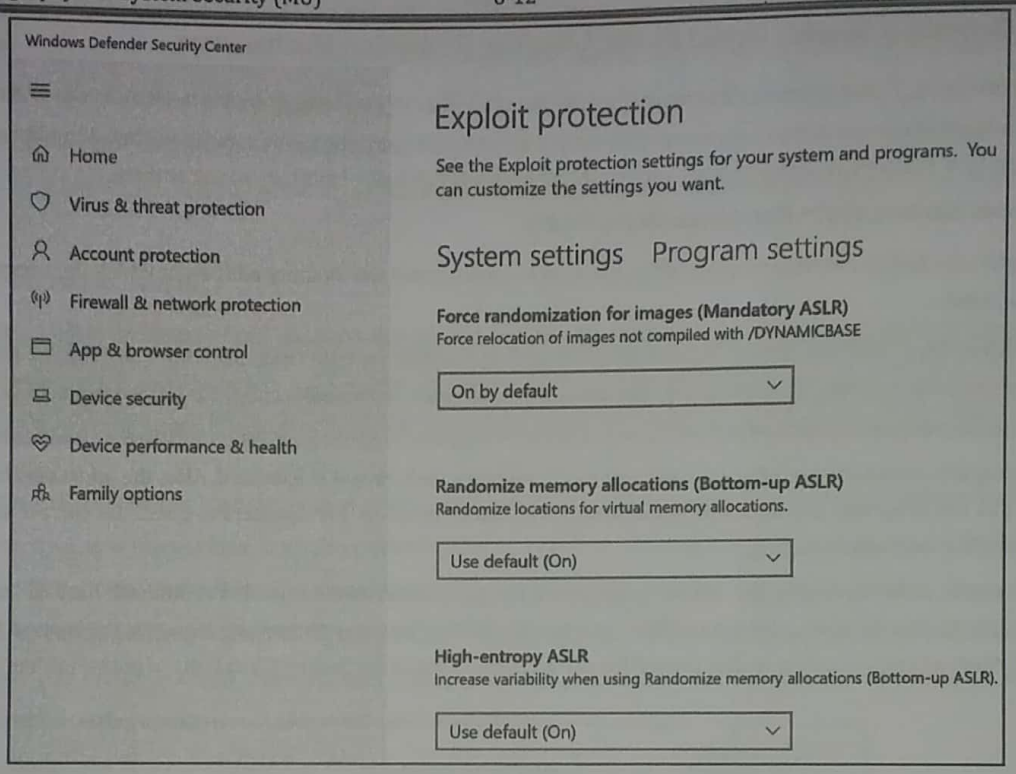
01370000	00001000	WinRAR		PE header
01371000	0010E000	WinRAR	.text	code
0147F000	0001A000	WinRAR	.rdata	imports
01499000	000A1000	WinRAR	.data	data
0153A000	00001000	WinRAR	.gfids	
0153B000	00001000	WinRAR	.tls	
0153C000	00037000	WinRAR	.rsrc	resources
01573000	0000D000	WinRAR	.reloc	relocations

ASLR is set at the operating system level and the programs supporting it can randomize their memory addresses on each execution.

On Linux, you can enable ASLR from command line as following.

```
# sysctl -w kernel.randomize_va_space=2→ Enable ASLR
# sysctl -w kernel.randomize_va_space=0→ Disable ASLR
```

On Windows®, you can enable ASLR from Windows® Security Settings.



2. Data Execution Prevention (DEP)

Definition : *Data Execution Prevention (DEP) is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system.*

Note here that DEP is a general feature that prevents the misuse of memory pages to run unauthorised code. It is not specific to buffer overflow but can also be used for protection from buffer overflow and hence listed here.

When using DEP, any program is allowed to run only from the certain authorised memory locations. If an attacker attempts to execute the program from an unauthorised memory location, such an attempt is denied.

For example, if the attacker tries to execute code from stack or heap memory segments, such an attempt is denied. Stack and heap are not meant for code execution but to temporarily store data.

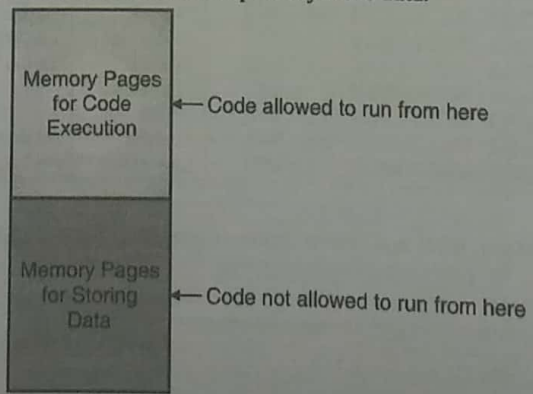


Fig. 6.2.5 : Memory pages marked as Data only

DEP can be implemented using,

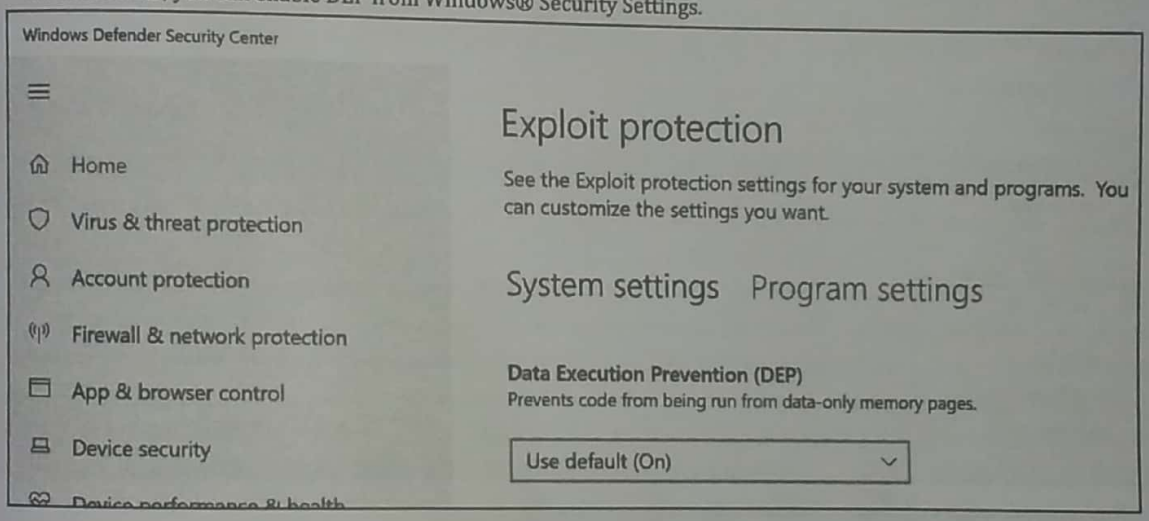
- The processor technology (hardware)
  - The no-execute page-protection (NX) processor feature as defined by AMD processors.
  - The Execute Disable Bit (XD) feature as defined by Intel processors.
- Software technology

Irrespective of what technology is used, the memory pages are mapped and marked for code execution or data only. DEP is set at the operating system level.

On Linux, you can enable DEP from command line as following.

```
# sysctl -w kernel.exec-shield=1 →Enable DEP
# sysctl -w kernel.exec-shield=0 →Disable DEP
```

On Windows®, you can enable DEP from Windows® Security Settings.



Advantages of OS-based buffer overflow protection mechanisms

1. The OS-based mechanisms do not require any program level changes.
2. OS takes the complete control over how memory is protected seamlessly.

Disadvantages of OS-based buffer overflow protection mechanisms

1. These mechanisms are architecture and platform specific and sometimes may not work equally across different operating systems.
2. It requires compatible processors that can enforce protection mechanisms as required by the operating system.
3. There could be system-wide performance impact.

6.2.2(B) Compiler Based Buffer Overflow Protection Mechanisms

The GNU Compiler Collection (GCC) is the compiler for C, C++ and several other languages. Overtime it has been updated with several compile time options (arguments and flags) using which you can adequately protect against buffer overflow. Your program needs to be compiled with the right compiler options to make use of such compile time buffer overflow protection mechanism.

Functions are a fundamental entity in programming languages. The assembler implementation of them has specific processor instructions. The two major instructions with respect to functions are the CALL (used for invoking the function) and the RET (used for going to the return address so that the rest of the program can continue to execute after the function completes its tasks).

A GCC compiled program, in order to call a function, pushes all the function parameters in the stack and issues the CALL instruction followed by the function address. When the function has to return to its caller it issues the RET instruction.

Before you learn about how various compiler-based protection mechanisms work, let's understand the memory layout of the stack with respect to the program execution process.

Consider a sample program.

```
char *func(char *msg)
{
        int var1;
        char buf[80];
        int var2;
        strcpy(buf,msg);
        return msg;
}
int main(int argv, char **argc)
{
        char *p;
        p = func(argc[1]);
        exit(0);
}
```

This program when executed is laid out on the stack as following :

func()	var2	4 bytes
func()	buf	80 bytes
func()	var1	4 bytes
func()	saved frame pointer	4 bytes
func()	return address	4 bytes
func()	func()'s arguments	4 bytes
main()	P	4 bytes
main()	saved frame pointer	4 bytes
main()	return address	4 bytes
main()	main()'s arguments	12 bytes

In the buffer overflow attack, the attacker tries to manipulate this stack layout such that to overwrite the return address of the function to her desired location so that her manipulated code can run and exploit the system.

func()	var2	4 bytes
func()	Overwrite to the desired address of the manipulated code	80 bytes
func()		4 bytes
func()		4 bytes
func()		4 bytes
func()	func()'s arguments	4 bytes
main()	P	4 bytes
main()	saved frame pointer	4 bytes
main()	return address	4 bytes
main()	main()'s arguments	12 bytes

Now that you understand how buffer overflow vulnerability is typically exploited, let's learn about the compiler-based protection mechanisms.

1. StackShield

✍ ***Definition :*** *StackShield is a compiler-based buffer overflow exploit protection technique.*

It works by copying the return address of the function to an area that cannot be overflown (for example, data segment).

When the function is invoked, its return address is copied to a safe region in the memory. When function is about to return, its return address from the stack is compared with the previously cloned return address (from safe memory region) when the function was invoked. If the two addresses match, it means that the return address on the stack was not modified (by buffer overflow) and the function successfully returns to the return address. If the two addresses don't match, the program terminates and does not let the attacker exploit the buffer overflow she created. It does not let the function return to the manipulated return address thus avoiding the exploit from buffer overflow.

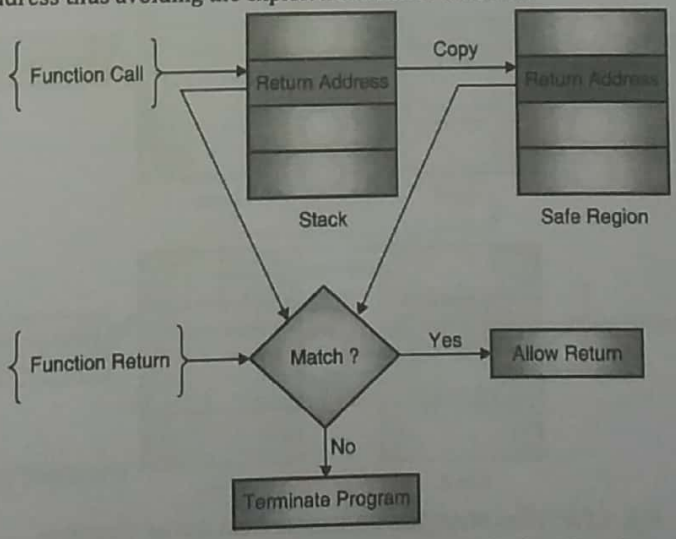


Fig. 6.2.6 : How StackShield protects from Buffer Overflow

2. StackGuard

✍ ***Definition :*** *StackGuard is a compiler-based buffer overflow exploit protection technique.*

It works by inserting a canary (random value) next to the return address. If the buffer overflow exploit is carried out, it would need to overwrite the canary value. Before jumping to the return address, the canary value written during the function call is checked. If it is not modified, the return is allowed else the program is terminated.

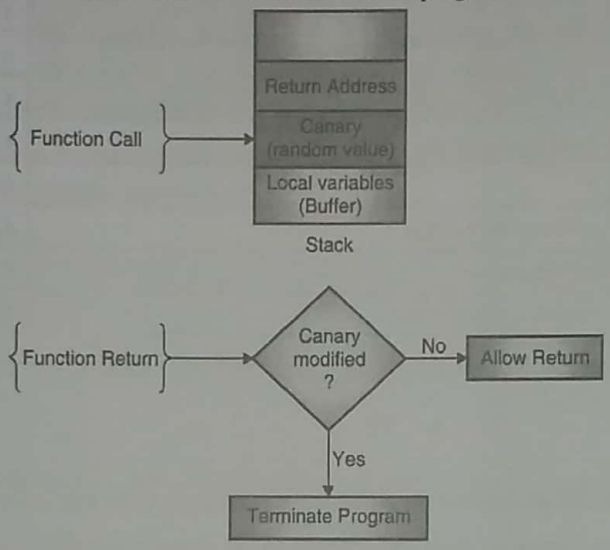


Fig. 6.2.7 : How StackGuard protects from Buffer Overflow

3. MemGuard

Definition : *MemGuard is a compiler-based buffer overflow exploit protection technique.*

It works by marking the return address as non-writable during the lifetime of the function. Any write attempt to this protected region during the function's active lifetime results in exception. The program is terminated, and the buffer overflow exploit is avoided.

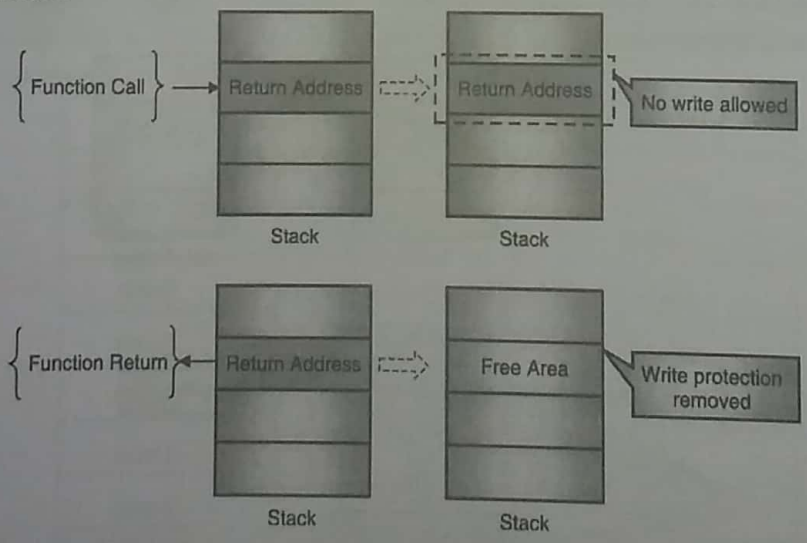


Fig. 6.2.8 : How MemGuard protects from Buffer Overflow

4. Other gcc patches and compiler options

Overtime there have been several additions to the gcc compiler options (flags and arguments to compile with) for safeguarding against the buffer overflow technique. They are listed here for your reference.

- -fstack-protector-strong argument
- /GS switch for Microsoft Visual Studio
- -qstackprotect – IBM compiler

Advantages of Compiler-based buffer overflow protection mechanisms

1. Since the protection mechanism is directly applied at the program level, there is no system-wide performance impact.
2. It does not depend upon the architecture or processor capabilities.
3. It does not require program level changes. Just compiling it with the right compiler options provides protection.

Disadvantages of Compiler-based buffer overflow protection mechanisms

1. You require source code to compile it with the right compiler options. If you do not have access to the source code, protection cannot be provided.
2. Each program and library need to be compiled with the right compiler options to provide overall protection from buffer overflow.

6.3 Types of Buffer Overflow

There could be several types of buffer overflow attacks that could be potentially carried out. They are classified as the following based on the area of memory they exploit or based on the type of exploit.

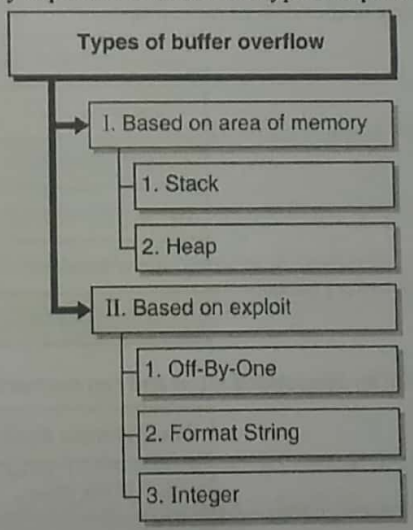


Fig. 6.3.1 : Types of Buffer Overflow

6.3.1 Based on Area of Memory

6.3.1(A) Stack Overflow

Stack overflow is the most common type of buffer overflow technique. You have learnt about it in the previous sections on how the stack can be manipulated by overwriting the return address and returning the functional pointer to an address from where the attacker can execute the malicious code. This is also called stack smashing.

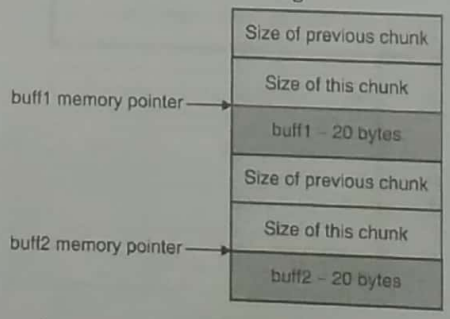
6.3.1(B) Heap Overflow

A heap overflow or heap overrun exploits the heap area. As you know, the memory on the heap area is dynamically allocated by your program at runtime. For example, when you use *malloc* or the *C++ new* operator, the memory is assigned from the heap area. It typically saves the program data and its size is not known previously.

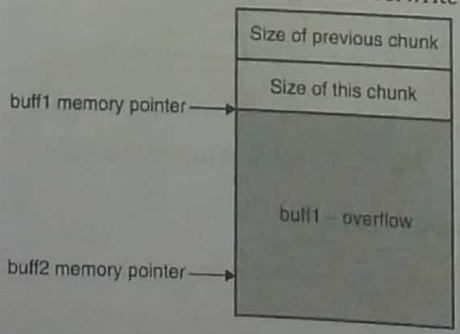
Heap overflow is carried out by corrupting the data in such a way so as to cause overwrites in the other areas of the heap. The program could crash, or the data could be generally corrupted. Let's see an example.

```
int main(void)
{
  char *buff1, *buff2;
  buff1 = malloc(20);
  buff2 = malloc(20);
  gets(buff1);
  free(buff1);
  exit(0);
}
```

For the given program, the heap memory is allocated as following.



In the example, the *gets(buff1)* can take an unbounded input and can overwrite *buff2* space in the heap as well.



Then, the function *free(buff1)* frees not only *buff1* space but also *buff2* space.

Size of previous chunk
Size of this chunk
Free Space
Free Space
Free Space
Free Space

Now, if you try calling *buff2*, it might crash the program or make it work in an unexpected way.

6.3.2 Based on Exploit

6.3.2(A) Off-By-One

This type of overflow is usually caused due to the programmer's oversight. The array element definition usually starts from 0 and while addressing the elements in loops, you might refer to an array element which is outside the definition.

Let's take an example.

```
void func(char *str)
{
  char buff[128];
  int i;

  for (i=0;i<=128;i++)
    buff[i] = str[i];
}
int main(int argc, char **argv)
{
func(argv[1]);
}
```

If you look at the example code carefully, you see <= operator in the *for* loop. The array definition is for 128 elements but the *for loop* starting from element 0 and going until 128 would make it 129 elements. This would cause overflow by one byte.

6.3.2(B) Format String

Languages such as C provide formatting parameters that can be effectively used to process variables as per the required formatting. Some of these parameters are as mentioned in the Table 6.3.1.

Table 6.3.1 : Format String Parameters

Parameters	Output	Passed as
%p	External representation of a pointer to void	Reference
%x	Read data from stack (hexadecimal)	Value
%s	Read character strings	Reference
%n	Writes the number of characters into a pointer	Reference

In the Format String exploit, the attacker passes the specialized formatted input values such as to read the stack, write to the stack or cause a system crash. Let's see an example.

```
#include <stdio.h>
void main(void)
{
  char a[10];
  gets(a);
printf(a);
}
```

1. Crashing the program

The attacker may provide input as "%s%s%s%s%s".

- For each %s, *printf()* will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character is encountered.
- If the number fetched by *printf()* is not an address, the memory pointed by this number might not exist and the program will crash.

```
%s%s%s%s%s
Segmentation fault
```

2. Viewing the stack

The attacker may provide input as "%x %x %x %x %x"

For each %x, *printf()* will display the stack content in the hexadecimal format.

```
%x %x %x %x %x
bcf81011 bcd5d9f0 fbad2288 bcf81012 0
```

3. Writing to the stack

Let's take a different example.

```
#include <stdio.h>
void main(void)
{
  int test=5;
  char buff[10];
  gets(buff);
printf("Original value of test:%d\n",test);
printf(buff,&test);
printf("\nNew value of test:%d\n",test);
}
```

In this example, the statement *printf(buff,&test)* can overwrite the value of *test*. Let's see a malicious input.

Assume the attacker enters "iamahacker%n" as input for *buff*. The *%n* format parameter counts the number of characters before it and writes the count of characters to the memory location of *test*. The value of *test* is thus modified, and the attacker has successfully written to the stack exploiting the *%n* format parameter.

```
iamahacker%n
Original value of test:5
Iamahacker
New value of test:10
```

6.3.2(C) Integer

The integer overflow can occur when you take integer inputs but their processing results into larger values than the variable can hold.

Let's see an example.

```
#include <stdio.h>

void main(void)

{

  unsigned short i = 60000;

  unsigned short j = 50000;

  unsigned short result = i * j;

printf ("%d",result);

}
```

Unsigned short data type can hold 2 bytes up to the numbers 0 to 65,535. In the given example, both the variables are under the limit of what unsigned short can hold but the result of their multiplication is too large to be accommodated in an unsigned short variable. This results in memory fault and incorrect result.

```
24064

...Program finished with exit code 5
```

6.4 SQL Injection (SQLi)

Q. Explain briefly with example, how the SQL injection attack. **MU - Dec. 17, 3 Marks**

✍ ***Definition :*** *SQL Injection (SQLi) is a vulnerability exploiting which an attacker can inject malicious SQL commands and cause the database server to execute them.*

Using SQLi, the attacker can,

- Read sensitive and unauthorised data from the database,
- Modify data,
- Execute administrative commands on the database,
- And can also Issue commands to the operating system running the database,

Note here that the presence of SQLi attack does not mean that the SQL databases are inherently vulnerable or exploitable directly. This exploit only works if the application developer has programmed the application in such a way that the attack is possible. SQL databases just behave as they should. So, even if the attack is called SQLi, the fix for this exploit is not in the SQL databases. The fix is required in the application developer's code as we will see later on.

SQLi is proven to be one of the most dangerous and impactful exploits on database-based applications. It has maintained the first position in the OWASP's list of Top Ten Most Critical Application Security Risks from several years.

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

6.4.1 How does SQLi Work?

Q. With the help of examples explain non malicious programming errors. **MU - Dec. 15, 5 Marks**

There could be several non-malicious programming errors such as buffer overflow, SQL Injection, Cross-site scripting etc. that can be exploited by attacker. Let's take an example of SQL Injection (SQLi).

SQLi can potentially be exploited where the user input is directly used to form SQL database queries. Instead of providing the expected input, the attacker can provide malicious input such that the original and desired SQL query is changed. The attacker can control the behaviour of the SQL queries and can manipulate what SQL queries are run.

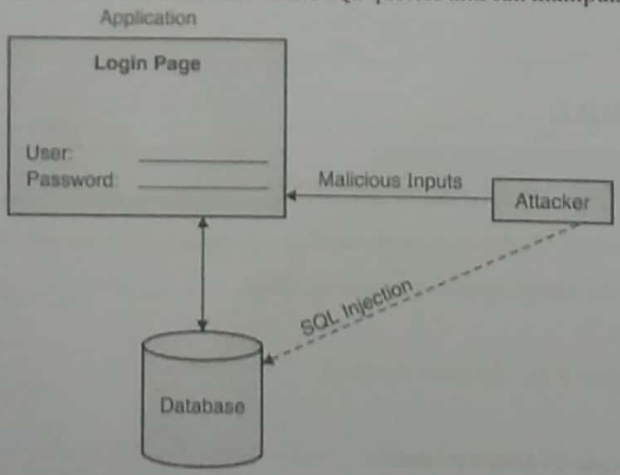


Fig. 6.4.1 : SQL Injection

Let's see an example.

```
query = "SELECT * FROM users WHERE name = ' + userName + ';"
```

This query takes *userName* from the user. If the user passes a valid input such as Joe as the *userName*, the query statement is correctly formed as following.

```
query = "SELECT * FROM users WHERE name = ' + Joe + ';"
query = "SELECT * FROM users WHERE name = 'Joe';"
```

But what if the user is malicious and passes *'OR '1'='1* as input?

```
query = "SELECT * FROM users WHERE name = ' + ' OR '1'='1 + ';"
query = "SELECT * FROM users WHERE name = " OR '1'='1';"
```

So, what happened here?

The *name* variable is searched for blank (' ') and then there is an *OR* statement which will always evaluate to True. ('1'='1'). So, overall, the SQL query formed will always run and print the entire *users* table!

So, the intention of the developer was to get the user name from the user and populate that value to form the dynamic query which can then show that particular user's details. But the malicious input provided by the user resulted in forming a query that would print the entire table and disclose the details of all the users contained in the database.

Let's consider some other malicious inputs.

```
' OR '1'='1 --
' OR '1'='1 {
' OR '1'='1 /*
```

In SQL, you can provide comments using -- (double hyphen), { } (braces) or /*...*/ (C-style multi-line comments). An attacker can pass injection statements followed by comments. Then the rest of the query is not executed, and the attacker can only execute the part of the query she wishes to execute.

Let's take the following query as an example.

```
query = "SELECT * FROM users WHERE
        username = ' + username + '
        AND password = ' + password + ';"
```

In this example, the system is trying to authenticate the user by taking the user name and the corresponding password. So, if the user name is valid and the password entered is also valid, the query will return the result else it will not return any results.

Now, an attacker can pass the input as *' OR '1'='1';"* -- for *username* and any random value for *password*.

```
query = "SELECT * FROM users WHERE
        username = " OR '1'='1';" -- '
        AND password = ' + password + ';"
```

Such a malicious input comments out the password check in the SQL query and returns True. The user is successfully authenticated bypassing providing any user name or password.

The attacker can also add more SQL queries as malicious input. For example, consider the following query.

```
query = "SELECT * FROM users WHERE
        username = " OR '1'='1'; DROP table users;" -- '
        AND password = ' + password + ';"
```

The malicious input of *' OR '1'='1'; DROP table users ; "* -- can now not only authenticate the user but also can delete the table containing the user information.

6.4.2 Types of SQLi

There are several variants of SQLi.

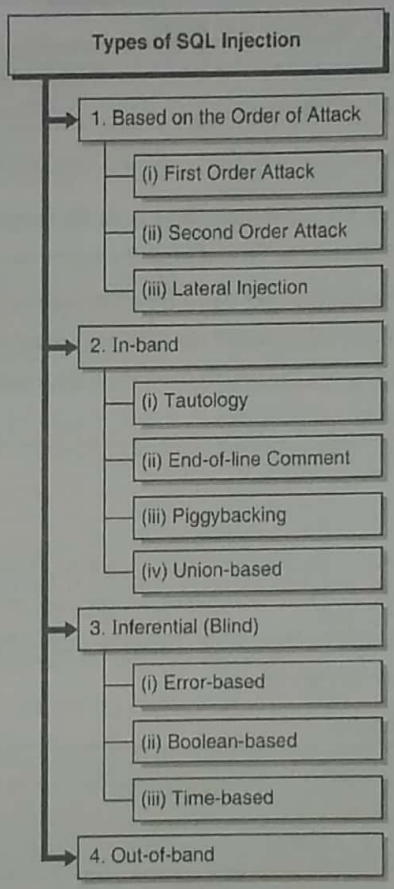


Fig. 6.4.2 : Types of SQL injection

6.4.2(A) Based on the Order of Attack

1. First order attack

✍ ***Definition :*** *In the first order attack, the attacker enters a malicious input and causes the SQLi attack immediately.*

The injected malicious SQL query is run immediately, and the attackers gets the results. The SQLi examples that you have learnt in the previous section are all first order attacks. A single query is manipulated, and the results are immediately obtained.

2. Second order attack

✍ ***Definition :*** *In the second order attack, the attacker first injects the malicious query (or data) into the database and then gets that malicious query (or use the malicious data) subsequently executed by another activity directly from the database.*

The result of second order SQLi attack is not obtainable immediately. There might be some time elapsed between the time at which the malicious query (or the malicious data) is injected and the time when the malicious query is actually executed (or malicious data is actually used), and the result of the attack is seen.

Let's see an example.

```
INSERT INTO users(username, password) VALUES("admin--", "random");
UPDATE users SET password = '1' WHERE username = admin--
```

Usually there are two SQLi attacks done.

First, the attacker manages to insert a new user into the database (using the first order attack that you learnt earlier or by just following the process of creating a new account on the system). Note here that the user name added is specifically having -- at the end of it, but since it is passed inside " ", the SQL query will accept that as the value and insert it into the database without complaining.

In the second statement, the attacker wishes to reset her password for *admin--* account she created. She passes *admin--* as the user name whose password needs to be reset. The database executes the query but due to -- at the end of *admin--*, it actually resets the password of the original *admin user*. Now, since the password of the original admin user is reset, the attacker can login as the admin on the system.

So, as you learnt in the example, there were two parts to the SQLi. First a value was stored in the database and then that stored value was exploited to carry out the SQLi.

3. Lateral Injection

✍ ***Definition :*** *In Lateral Injection, the attacker modifies the system functions to inject malicious query.*

Using Lateral SQL Injection, an attacker can exploit stored procedure (SQL functions). For example, the function *TO_CHAR()* can be manipulated by changing the environment variables NLS_Date_Formator NLS_Numeric_Characters. You can include any text in the format model and can exploit SQLi.

6.4.2(B) In-band

1. Tautology

Tautology means a statement that is always true. For example, "a beginner who has just started" is a tautology.

✍ ***Definition :*** *Tautology is a form of SQLi where conditional statements are injected that always evaluate to true.*

Following is a simple example of a tautology.

```
SELECT * FROM users WHERE username='' or 1=1 --
```

The use of 1 = 1 -- makes the entire *WHERE* condition as a tautology (always evaluating to true).

2. End-of-line Comment

✍ ***Definition :*** *End-of-line comment is a form of SQLi where comments are used to terminate the SQL query prematurely (inappropriately) to run only a part of the query.*

Following is a simple example.

```
SELECT * FROM users WHERE username='admin'-- AND password = adminpasswd
```

The use of -- terminates the query prematurely and the query thus returns the entire details of the *admin user* without requiring the *admin* password.

3. Piggybacking

Piggybacking means to ride on someone's shoulder.

✍ ***Definition :*** *Piggybacking is a form of SQLi where additional queries are injected to run with the original query.*

Following is a simple example.

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1'; DROP table users;
```

The use of ; after the first query allows to add the second query. When the first query completes, the database server runs the second query followed by the ; (*DROP table users*). The attacker can add as many queries as she wishes separated by ;.

4. Union-based

The UNION operator is used to combine the result-set of two or more SELECT statements.

✍ ***Definition :*** *Union-based is a form of SQLi where the UNION operator is used to inject the second query to the original query.*

Following is a simple example.

```
SELECT username FROM users where username = ' ' OR '1'='1'
UNION SELECT empname FROM employees
```

There are two tables *users* and *employees*, and both have a field containing names. The attacker can use the *UNION* operator to reveal the employee names from the *employees* table.

6.4.2(C) Inferential (blind)

1. Error-based

✍ ***Definition :*** *Error-based is a form of SQLi where the attacker causes errors to learn about the environment.*

For gathering information about the environment such as database type, version, OS, table names, etc. the attacker might cause system errors by injecting random queries. If those errors are not sufficiently handled in the application, then those errors would be visible in the application. The attacker can use the gathered information to launch more sophisticated and known attacks.

Following is a simple example.

```
SELECT * FROM users WHERE username= ' ' AND pin=convert(int,(select top 1 name from sysobjects where xtype='u'))
```

Here the attacker purposely creates an error by trying to convert name into integer. *sysobjects* is a system table that stores all the table names present on the database server. The attacker tries to check the table names starting from first entry *(top 1)* in the *sysobjects* table. The attacker is only interested in the tables created by the user *(xtype='u')*.

The query when executed gives the following error.

```
Microsoft OLE DB Provider for SQL Server (0x80040E07)
Error converting nvarchar value 'CreditCards' to a column of data type int
```

The attacker gets to know two things from this error :

- The database server running behind the application is Microsoft® SQL Server®
- There is a table by name CreditCards present on the database server.

The attacker can then use this information to launch further attacks.

2. Boolean-based

✍ ***Definition :*** *Boolean-based is a form of SQLi where the attacker injects true/false statements to learn about the environment.*

Following is a simple example.

Suppose the attacker injects the condition such that it always evaluates to true.

```
SELECT * FROM users WHERE username='' or 1=1 -- AND password = mypasswd
```

Suppose the response from the application is *"Invalid Password"*.

The attacker can infer (guess) two things from it:

- Either her malicious input was detected and rejected.
- Or the check for password is a separate query (because -- would have commented out that password check).

Now, suppose the attacker changes the condition such that it always evaluates to false.

```
SELECT * FROM users WHERE username='' or 1=0 -- AND password = mypasswd
```

Suppose the response from the application is *"Invalid Username and Password"*.

The attacker learns that the response is different from the previous one when she inserted a true condition. This would conclude that the field *username* is injectable and could be exploited.

3. Time-based

✍ ***Definition :*** *Time-based is a form of SQLi where the attacker injects random queries and evaluates the time it takes to run the query and get the result to learn about the environment.*

Time-based SQLi injects the SQL query to the database such that it forces the database to wait for a specified amount of time (in seconds) before responding. The response time indicates to the attacker whether the result of the query is true or false.

Following is a simple example.

```
SELECT accounts FROM users WHERE login='abc' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ' AND
pass='' AND pin=0
```

In this example, the attacker waits for 5 seconds if the ASCII code of the first character from the first table name in *sysobjects* is greater than the ASCII code of the character X. If the ASCII code is greater, then there would be a 5 second delay and if the ASCII code is not greater, then there would not be a 5 second delay.

6.4.2(D) Out-of-band

✍ ***Definition :*** *Out-of-band is a form of SQLi where the attacker injects the SQL query via one medium and receives the results via another medium.*

It is not very common because it depends on the features enabled on the database server such as the ability to send emails or notifications. Out-of-band techniques offer the attacker alternatives for evaluating attack results.

6.4.3 Preventing SQLi

Some of the ways in which SQLi can be prevented are as following.

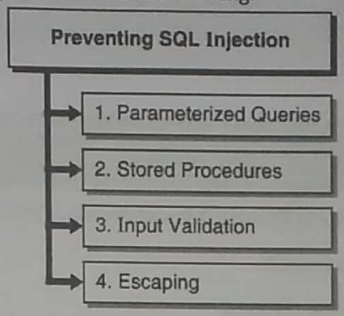


Fig. 6.4.3 : Preventing SQL Injection

1. Parameterized Queries

The primary cause of SQLi exploit is that the user input is directly used to form the SQL query.

✍ ***Definition :*** *In parameterized queries, the user input is separated from the SQL query such that the input is taken literally and not as part of the SQL query.*

Let's see an example.

Assume a SQL query that is vulnerable to SQLi.

```
SELECT * FROM users WHERE username= "' + name + "'
```

Here the variable *name* is part of the query. This allows the attacker to provide malicious input for the *name* variable and that malicious input becomes the part of the query and the SQLi gets exploited.

Instead of writing the query this way, parameterized queries separate the input variable from the actual query.

```
paraName = getRequestString(name);
paraQuery = "SELECT * FROM users WHERE username=@0";
db.Execute(paraQuery, paraName);
```

The database server then takes the variable input literally instead of part of the SQL query. So, even if the attacker provides a malicious input, the database server would not execute it. The *@0* provides the argument number that should be taken as input for executing the query in the *db.Execute* function call.

Note : The example given previously is as per ASP.NET language. Each language has its own way of constructing parameterized queries. For example, following is an example of creating parameterized query in Java.

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

2. Stored Procedures

✍ ***Definition :*** *A stored procedure is like a function or a subroutine that you can save in your database and call and execute multiple times.*

For example, following is a stored procedure that does not require a parameter.

```
CREATE PROCEDURE ShowAllStudents
AS
SELECT * FROM students
GO;

EXEC ShowAllStudents;
```

Following is a stored procedure that takes one parameter.

```
CREATE PROCEDURE ShowAllStudents @Class nvarchar(30)
AS
SELECT * FROM students WHERE Class = @Class
GO;

EXEC ShowAllStudents Class = "BE";
```

Following is a stored procedure that takes multiple parameters.

```
CREATE PROCEDURE ShowAllStudents @Class nvarchar(30), @Dept nvarchar(30)
AS
SELECT * FROM students WHERE Class = @Class AND Dept = @Dept
GO;
EXEC ShowAllStudents Class = "BE", Dept = "Computer";
```

The idea behind using stored procedures is to separate out the query from the input data. It works very similar to the parameterized queries and can be effectively used to prevent SQLi.

3. Input Validation

SQLi exploit is possible because the user input is not adequately validated for correctness. As a developer, you should evaluate the user input before accepting or processing it. Lookout for things like,

- Length of the input
- Format of the input
- Any special characters that you don't believe could be a legitimate value

4. Escaping

As you learnt in XSS, you can escape (replace) user input with the respective codes. In this technique, all the user input is escaped before putting it in a query. It varies from database to database in its implementation. When you escape user input, it is treated literally, and the database server does not treat it as part of SQL query even if the attacker desires so.

For example, following are some escape sequences for MySQL database server.

Escape Sequence	Character Represented by Sequence
\0	An ASCII NUL (X'00') character
\'	A single quote (') character
\"	A double quote (") character
\b	A backspace character
\n	A newline (linefeed) character

Escape Sequence	Character Represented by Sequence
\r	A carriage return character
\t	A tab character
\Z	ASCII 26 (Control+Z); see note following the table
\\	A backslash (\) character
\%	A % character; see note following the table
_	A _ character; see note following the table

This is an old technique and should not be used quite often. It can be complex and error-prone.

6.5 Malware (or Malicious Code)

Malware (or Malicious code) refers to,

✍ ***Definition** : The category of programs that infect the information systems and are motivated to gain unauthorised access to the system, information contained therein or to fulfil any other malicious objectives.*

These are usually spread through emails, social media, malicious websites, documents and other attachments. You should be very careful while clicking on a link or opening an email from a stranger or downloading an interesting attachment. These all could be mechanisms to infect your system.

6.5.1 Concept Building-Activities related to Malware

Before you understand various types of malware, let's understand various activities associated with malware. A malware may exhibit some of these activities or all of them.

1. **Installation :** How the malware reaches the system, e.g. from attachment.
2. **Detection and removal :** How the malware's presence can be detected, e.g. anti-virus.
3. **Payload :** Actual function that the malware performs, e.g. delete system files.
4. **Trigger :** Event that invokes (or activates) the malware, e.g. clicking a file.
5. **Replication :** Capability of the malware to further copy itself and infect other systems.
6. **Eradication :** The malware might remove itself after delivering the payload.

6.5.2 Types of Malware

Let's learn about some of the common forms of *malware (or malicious code).*

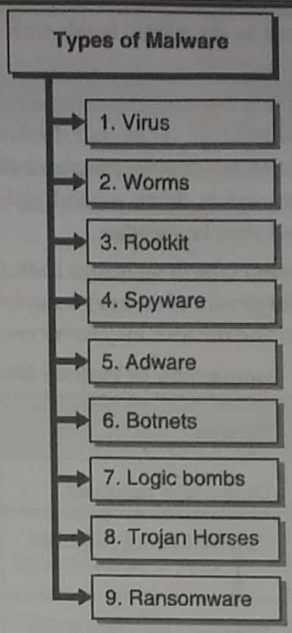


Fig. 6.5.1 : Types of Malware

1. **Virus :** Virus is a category of malicious code that cannot self-replicate itself to deliver the payload. It requires a trigger (user action) to activate and then deliver the payload (harm the system in a particular way). It could deliver various payloads such as deleting system files, displaying specific messages or stealing information. ILOVEYOU and Melissa are two well-known viruses.
2. **Worms :** Worms are smaller malicious programs that can self-replicate and trigger without needing a user action. They are targeted towards infecting the systems and delivering the required payload of their own. One of the most common examples of worm is Stuxnet which was used to damage Iran's uranium infrastructure.
3. **Rootkit :** Once the attacker has access to the system, she can upload a set of tools that can remain on the system for her future access and further exploiting the system by becoming administrator (elevation of privilege attack). Such set of tools are called rootkits. Rootkits allow access without the attacker having to redo the attack to gain access to the system.
4. **Spyware :** Spyware is a category of malicious programs which are secretly downloaded (say when you are browsing a website) and installed on the system for the purpose of gaining access to the sensitive information such as passwords. These can be further used to take control over your system and install other malicious programs.
5. **Adware :** Adware is a special category of malicious programs that are intended to serve advertisements through banners, pop-ups or screen messages. They do not perform a malicious activity *per se but* could be very annoying. The purpose of adware is to drive sales through continuous ads.
6. **Botnets :** Bot is a short word for 'robot'. Botnets are a huge network of malicious programs (bots) that do various tasks for its master. These are typically used in the web-based environments where the master uses these automatic programs to carry out malicious actions. For example, botnets can send thousands of spam emails to millions of people every day.
7. **Logic bombs :** Logic bombs are malicious programs that are automatically triggered when the desired logic is met. For example, an attacker can set the program to delete all files when it becomes 12.12.12.12.12.12 (12 Dec. 2012 at 12th hour, 12th Minute and 12th Second). Logic bombs can also be used like a time bomb.

For example, the administrator may choose to set a logic bomb such that if she does not deactivate it before a time, automatically the system is infected.

8. **Trojan Horses :** These are malicious programs that hide themselves in legitimate (right) looking programs.
For example, a trojan might be called notepad.exe. When you double click it, it might call the original notepad.exe program but also do several other things such as sending sensitive information to the attacker's machine. You believe that the program you clicked is legitimate and is doing a good and desired action, but it is actually carrying out malicious activities behind the scene without your knowledge.
9. **Ransomware :** Ransomware is the most recent type of malicious code. In this, the attacker takes over the user's system and locks it out and demands a ransom (a large sum of money) for unlocking it. If the user does not pay the money, the attacker usually erases the complete system and the user may never recover the system and data that was on it.

Table 6.5.1 : Comparison of Various Malicious Code

Type of malware	Purpose	Trigger	Replication	Countermeasure
Virus	Deliver payload	User action required	User action required	Anti-virus
Worms	Deliver payload	User action not required	User action not required	Anti-virus
Rootkits	Future access	Attacker uses these tools	Not applicable	Anti-virus
Spyware	Gather sensitive information	User action not required	Not applicable	Anti-malware
Adware	Serve advertisements	User action not required	Not applicable	Anti-malware
Botnets	Deliver payload	By botnet master	In thousands to form a network	Anti-malware
Logic bombs	Deliver payload	User action not required	Not applicable	Separation of duties
Trojan Horse	Deliver payload	User action not required	Not applicable	Anti-virus
Ransomware	Extort money	By the attacker	Not applicable	Backup of data

Now that you have an overall idea of malware, let's dive deeper into some of them.

6.5.2(A) Virus and Worms

Q. Write a brief on - Viruses and their types. MU - Dec. 15, 5 Marks

Q. What are the different types of viruses and worms? How do they propagate? MU - Dec. 16, 10 Marks

What is a virus?

Definition : *Virus is a category of malicious code that cannot self-replicate itself to deliver the payload.*

It requires a trigger (user action) to activate and then deliver the payload (harm the system in a particular way). It could deliver various payloads such as deleting system files, displaying specific messages or stealing information. ILOVEYOU and Melissa are two well-known viruses.

What is a worm?

✍ ***Definition :*** *Worms are smaller malicious programs that can self-replicate and trigger without needing a user action.*

They are targeted towards infecting the systems and delivering the required payload of their own. One of the most common examples of worm is Stuxnet which was used to damage Iran's uranium infrastructure.

Types of virus

Viruses are usually classified based on :

1. How they live in memory
2. How they spread
3. How they hide
4. What they deliver

Let us understand the classification first. We will then detail them out.

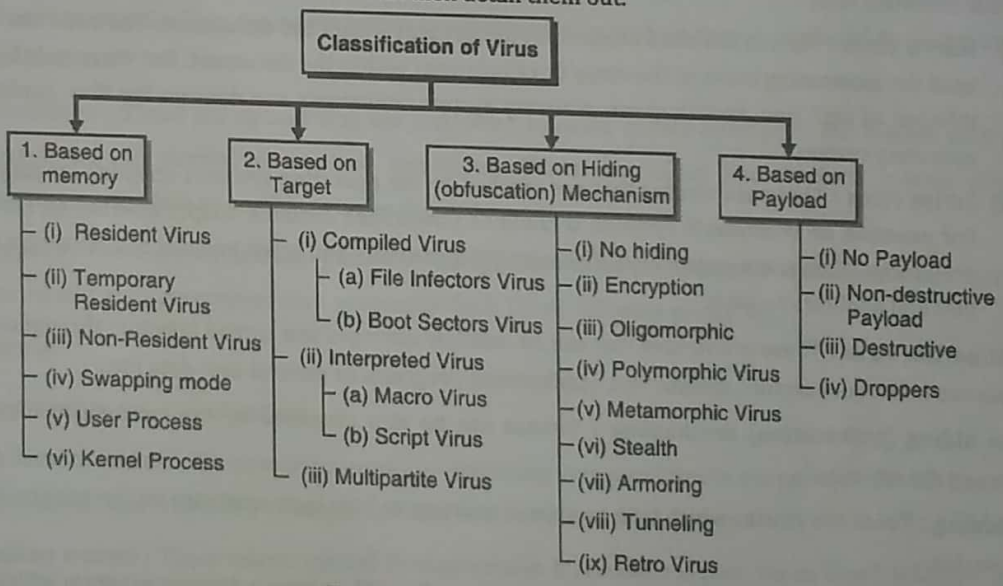


Fig. 6.5.2 : Classification of Virus

1. **Based on memory :** This classification depends upon how the virus stays within a system.
   - **(a) Resident virus :** These types of viruses stay in memory and infect any file or program that is opened.
   - **(b) Temporary resident virus :** These viruses stay in memory for some time and then vacate the memory based on time or after some event (say infecting 5 files).
   - **(c) Non-resident virus :** These viruses do not stay in memory. These infect the files by copying their payloads to the files directly. The relevant files are searched and then the payload is copied to them.
   - **(d) Swapping mode :** These viruses use swap space for residing. These switch from swap space to memory, infect a few files and return back to the swap space.
   - **(e) User process :** These viruses run with user level access and privileges. These can mostly infect only the user files without touching the system files (since for infecting system files, administrative level access is needed).

(f) **Kernel process :** These viruses are very dangerous as they get installed at the system level (say via device drivers or through system applications). These viruses run with the full administrative access and privileges and can infect any file, program or application on the system.

2. **Based on target :** This classification depends upon what the virus choses to infect (the target of infection).

(a) **Compiled virus :** These are viruses that run as machine executable. These are directly run by the operating system. The virus' source code is compiled to form an executable file. These can be further classified into,

(i) **File Infector virus :** File infectors corrupt the specific types of files. These often manipulate the file headers such that when the file is opened, it actually places the file pointer to the malware code execution point.

(ii) **Boot sector virus :** These viruses infect the boot sectors on the hard disk. Boot sectors are key partitions that help to power on and bring up the system for operation.

(b) **Interpreted virus :** The source code of the interpreted viruses is run at the execution timeline by line. Unlike compiled viruses, the interpreted viruses are run by the legitimate application programs when invoked. These are further classified into,

(i) **Macro virus :** Macros are code blocks that are present within the documents. The code blocks are usually used for automating some of the steps or calculations within the document. But these code blocks could be infected as well such that they run arbitrary system commands and damage the files, applications or the operating system.

(ii) **Script virus :** These are viruses that are invoked by the operating system's scripting language interpreter. For example, on Windows® systems, it could be PowerShell or Batch scripts whereas on Linux it could be shell script. Scripts are code blocks that exist independently and when invoked could damage the system or files and applications on it.

(c) **Multipartite virus :** These are viruses that can do multiple damages and spread infection throughout the system. These can infect boot sectors, system files, applications, programs or general user data files.

3. **Based on hiding (obfuscation) mechanism :** Viruses can be also classified by the mechanism they use to hide themselves on the system.

(a) **No hiding :** These are viruses which take no special measure to hide their existence on the system. Viruses rarely follow this.

(b) **Encryption :** These viruses use encryption to hide themselves. They have a decrypt program which exposes the virus when it's time to spread the infection on the system. Once the infection task is carried out, the virus again encrypts itself to hide.

(c) **Oligomorphic :** These are also called semi-polymorphic. Basically, these viruses use several decryption routines to hide itself as much as possible. With each infection, it changes the decryption routines to avoid getting detected by the anti-virus tools.

(d) **Polymorphic :** These viruses have some sort of mutation (changing) engine that changes the appearance of the viruses with each infection to make their detection very difficult. These make several changes to the encryption settings and also changes the decryption code accordingly.

(e) **Metamorphic :** These viruses change the virus body instead of appearance. It is done by using unneeded functions or commands or changing the order of virus code.

(f) **Stealth :** These viruses try to hide the infection by restoring the older properties of the file (time of modification, file size, etc.) after infection.

(g) **Armoring :** An armoring virus is a virus that makes analysis very difficult. These viruses use various techniques to hide themselves from detection and removal.

(h) **Tunnelling :** These viruses attach themselves to the system interrupts. Whenever there is an interrupt call, first the virus executes itself and then passes the control to the original interrupt.

(i) **Retro virus :** These viruses are specifically designed to bypass any security tools such as firewall, anti-virus or intrusion detection systems. The virus developers understand to a great extent how such security programs work and make it increasingly hard for these programs to detect viruses.

4. **Based on payload :** Based upon what the viruses deliver or do to the target system, they can be classified as under.

(a) **No payload :** These viruses do not affect systems as such or infect files. These are generally productivity killers that irritate by slowing down the system or creating minor nuisances.

(b) **Non-destructive payload :** These viruses as well are non-harming in nature except that they pop up repeated messages or graphic. They can control some hardware devices such as mouse, keyboard or cdrom and make them to malfunction.

(c) **Destructives :** These are viruses that are motivated to cause severe damage to the system such that they are damaged beyond normal operational conditions. These can affect boot sectors, system files, programs, applications, user data or anything else on the system.

(d) **Droppers :** These are viruses that just sit on the system and let the attacker use the system's resources to launch other attacks and cybercrimes such as identity theft, DDoS, software piracy, etc.

Types of worms

There are majorly two types of worms.

1. **Network Service worms :** These worms spread by exploiting vulnerabilities in the network service associated with an OS or application. Once it infects a system, it spreads to other networked systems.
2. **Mass mailing worms :** These worms spread through emails. It reaches a system via an email, infects it and then looks for other email addresses in the system to send a copy of itself to other users.

Table 6.5.2 : Comparison between Virus and Worm

Comparison Parameter	Virus	Worms
Trigger	User action required	User action not required
Replication	User action required	User action not required
Spread	Mostly remains on the infected system	Spreads throughout the network
Countermeasures	Antivirus	Antivirus, Firewall
Detection Complexity	High	Medium
Damage Caused	Usually high	Usually medium

6.5.3 General Guidelines for Preventing Malware

Some of the most common and general guidelines for preventing any form of malware are as following.

1. Regularly scan your systems using an adequate anti-malware program
2. Do not open emails and attachments from unknown sources
3. Control the urge to click on every URL that you see
4. Restrict visiting sites that support software piracy, promise free videos, music, books, etc.
5. Do not install programs from unknown or non-reputed sources
6. Regularly update your OS and installed applications to fix any known vulnerabilities
7. Do not plug external devices such as a pen drive from unknown sources
8. Limit physical access to the system

Review Questions

Here are a few review questions to help you gauge your understanding of this chapter. Try to attempt these questions and ensure that you can recall the points mentioned in the chapter.

[A] Software Vulnerabilities

Q. 1 With a simple code example, explain buffer overflow. **(6 Marks)**

Q. 2 You are reviewing code written by a colleague. She has used gets() in her code. What do you suggest and why? **(4 Marks)**

Q. 3 What can cause buffer overflow? **(7 Marks)**

Q. 4 Write a short note on ASLR. **(4 Marks)**

Q. 5 How Data Execution Prevention (DEP) can be used to provide protection from buffer overflow? **(5 Marks)**

Q. 6 Classify buffer overflow protection techniques. **(6 Marks)**

Q. 7 Julius is interested in buffer overflow protection. He comes to you asking about the advantages and disadvantages of OS-based and compiler-based protection mechanisms. What do you tell him? **(6 Marks)**

Q. 8 What is a canary? How and where it is used? **(6 Marks)**

Q. 9 Draw a block diagram of StackShield and explain. **(6 Marks)**

Q. 10 Explain any three types of buffer overflow. **(6 Marks)**

Q. 11 Write a short note on Format String vulnerability. **(4 Marks)**

Q. 12 Compare the types of XSS. **(6 Marks)**

Q. 13 Persisted XSS can exploit any visitor. Comment. **(6 Marks)**

Q. 14 Alex is worried about XSS. You tell him to relax and follow some guidelines on preventing XSS. He asks you which guidelines you would recommend. What do you tell him? **(6 Marks)**

Q. 15 How SQLi works? **(6 Marks)**

Q. 16 Classify SQLi. **(8 Marks)**

Q. 17 Write a short note on tautology. **(4 Marks)**

Q. 18 Database programmers are joining your company today. You are sent to explain company guidelines on preventing SQLi. What do you explain? **(7 Marks)**

[B] Malware

Q. 19 Compare various types of malware on the basis of Purpose, Trigger, Replication and suitable Countermeasure. **(6 Marks)**

Q. 20 Draw the virus classification diagram (no explanation required). **(6 Marks)**

Q. 21 Provide a comparison between a virus and a worm. **(8 Marks)**

Q. 22 Write a short note on Trojans (Trojan Horses). **(4 Marks)**

Q. 23 Write a short note on Logic Bombs. **(4 Marks)**

Q. 24 What are the components of bots? **(6 Marks)**

Q. 25 Describe various botnet architectures. **(8 Marks)**

Q. 26 Explain the attacks carried out by botnets. **(8 Marks)**

Q. 27 What is a rootkit? Explain its types. **(8 Marks)**

Q. 28 What are the ways of detecting rootkits? **(6 Marks)**

Q. 29 Ron wants to ensure that everyone in the company is aware of malware protection guidelines. You are required to make a presentation covering the malware protection guidelines. Which guidelines would you include in your presentation? **(4 Marks)**

❑❑❑