Buffer Overflow

Module 18

Ethical Hacking and Countermeasures v8

Module 18: Buffer Overflow

Exam 312-50

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Security News

Steam Gaming Platform Vulnerable to Remote Exploits; 50 Million at Risk

Source: http://threatpost.com

More than 50 million users of the Steam gaming and media distribution platform are at **risk for remote compromise** because of weaknesses in the platform's URL protocol handler, a pair of researchers at ReVuln wrote in a paper released this week.

Luigi Auriemma and Donato Ferrante discovered a number of **memory corruption** issues, including **buffer and heap overflows** that would allow an attacker to abuse the way the Steam client handles browser requests. Steam runs on Windows, Linux and Mac OSX.

The steam:// **URL protocol** is used to connect to game servers, load and uninstall games, backup files, run games and interact with news, profiles and download pages offered by Valve, the company that operates the platform. Attackers, Auriemma and Ferrante said, can abuse specific Steam commands via steam:// URLs to inject attacks and run other **malicious code** on victim machines.

"We proved that the current implementation of the **Steam Browser Protocol handling mechanism** is an excellent attack vector, which enables **attackers to exploit** local issues in a

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

remote fashion," Auriemma and Ferrante wrote. "Because of the big audience, the support for several different platforms and the amount of effort required to exploit bug via the Steam Browser Protocol commands, Steam can be considered a high-impact attack vector."

A large part of the problem rests in the fact that most browsers don't ask for **user permission** before interacting with the **Steam client**, and those that do, don't explain there could be a security issue. As a result, users could be **tricked into clicking** on a malicious steam:// URL or redirect browsers via JavaScript to a **malicious site**, the paper said.

The paper details five new remotely **exploitable vulnerabilities** in not only Steam, but also in the Source and Unreal game engines. Some of the games running on the affected platforms include Half-Life 2 Counter-Strike, Team Fortress 2, Left 4 Dead, Nuclear Dawn, Smashball and many others.

One of the more **dangerous** vulnerabilities discovered is involves the retailinstall command that allows Steam to install or restore backups from a local directory. An attacker can abuse the directory path to point to a remote network folder and then attack the function that processes a .tga splash image which is vulnerable to an integer overflow attack. A **heap-based overflow** results and an attacker could **remotely execute code**.

To exploit the Source game engine, Auriemma and Ferrante used **a malicious** .bat file placed in the startup folder of the user's account that executes upon the gamer's next login.

The pair also found several integer **overflow flaws** in the Unreal gaming engine by taking advantage of a condition where **Unreal** supports the loading of content from remote machines via Windows WebDAV or a SMB share. Malicious content could be **remotely injected** in this way.

Auto-update function vulnerabilities in a pair of games, All Points Bulletin and MicroVolts, were also discovered and exploited. The researchers were able to exploit a directory traversal to overwrite or create any malicious file.

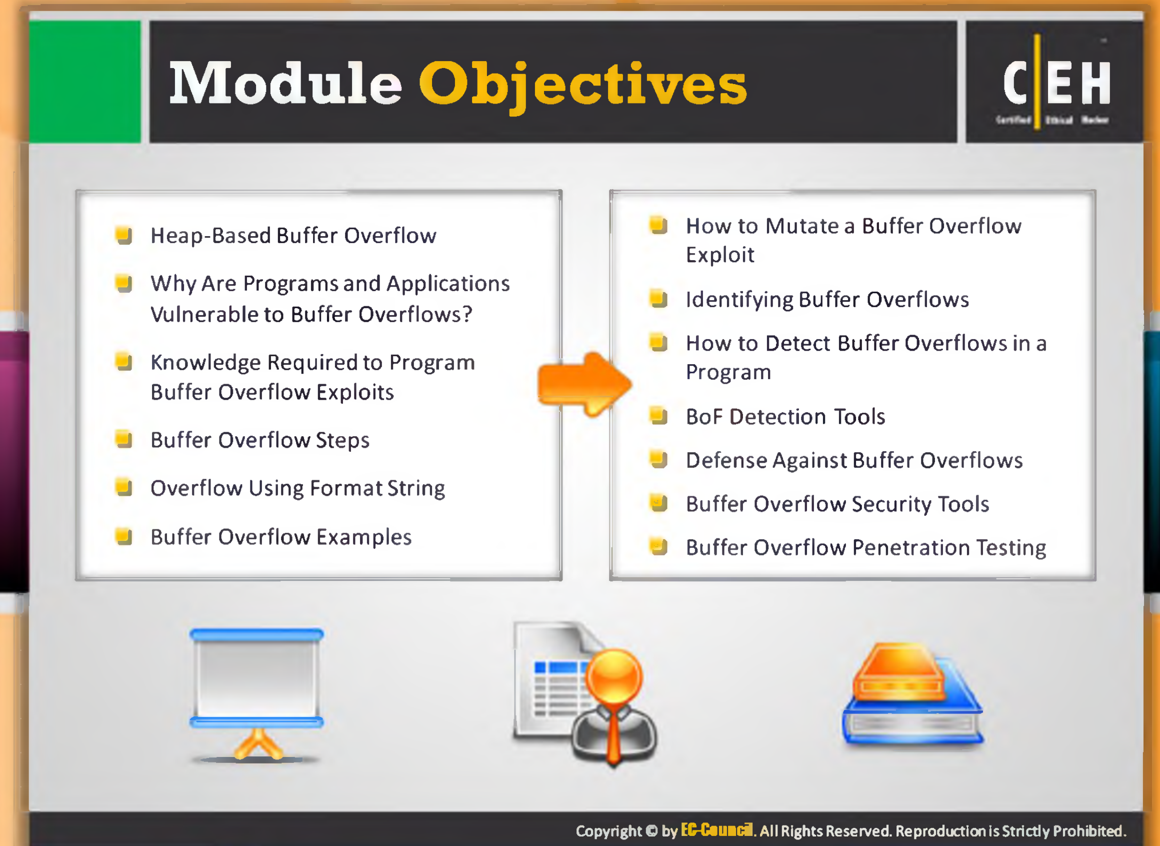
Users reduce the impact of these issues by disabling the steam:// URL handler or using a browser that doesn't allow direct execution of the Steam Browser Protocol. Steam could also deny the passing of **command-line arguments** to remote software.

Copyright © 2012 threatpost.com

By Michael Mimoso

http://threatpost.com/en_us/blogs/steam-gaming-platform-vulnerable-remote-exploits-50-million-risk-101912

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

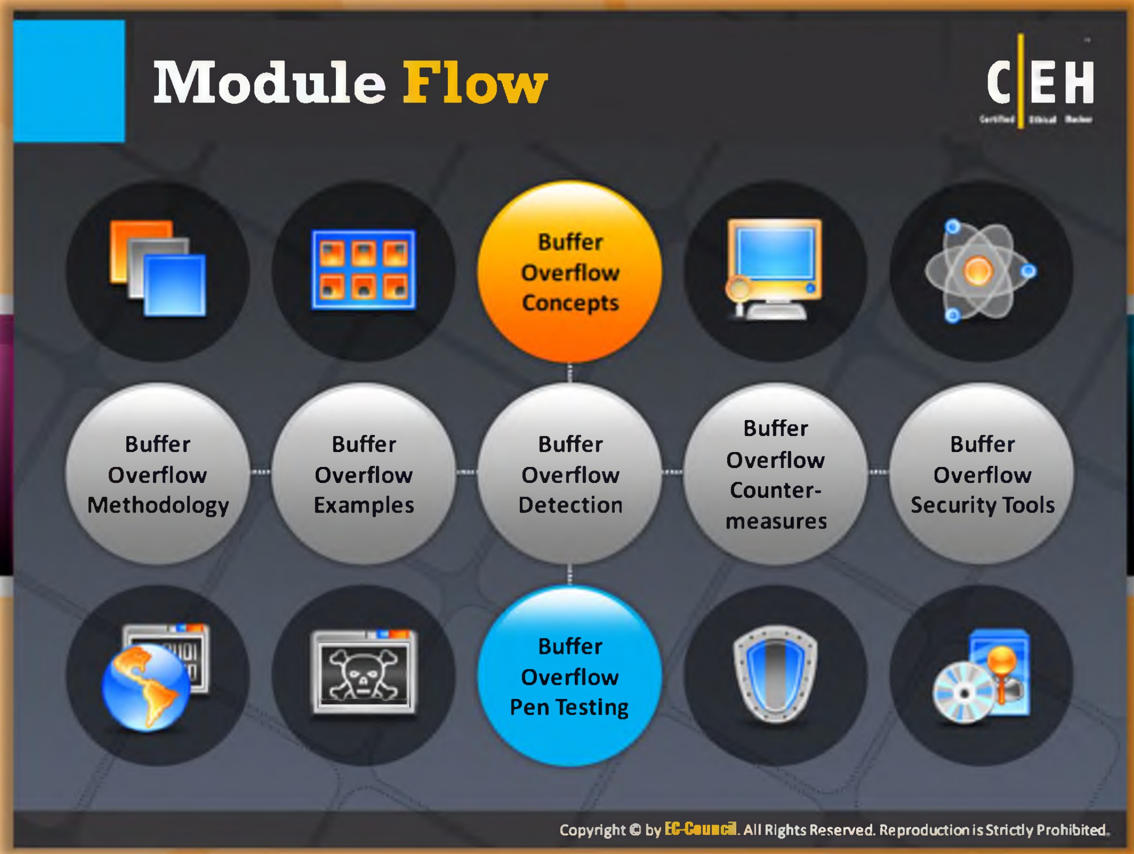


Module Objectives

Various security concerns, attack methods, and countermeasures have been discussed in the previous modules. **Buffer overflow attacks** have been a source of worry from time to time. This module looks at different aspects of buffer overflow **exploits** that include:

- Heap-Based Buffer Overflow
- Why Are Programs and Applications Vulnerable to Buffer Overflows?
- Knowledge Required to Program Buffer Overflow Exploits
- Buffer Overflow Steps
- Overflow Using Format String
- Buffer Overflow Examples
- How to Mutate a Buffer Overflow Exploit
- Identifying Buffer Overflows
- How to Detect Buffer Overflows in a Program
- BoF Detection Tools
- Defense Against Buffer Overflows
- Buffer Overflow Security Tools
- Buffer Overflow Penetration Testing

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Module Flow

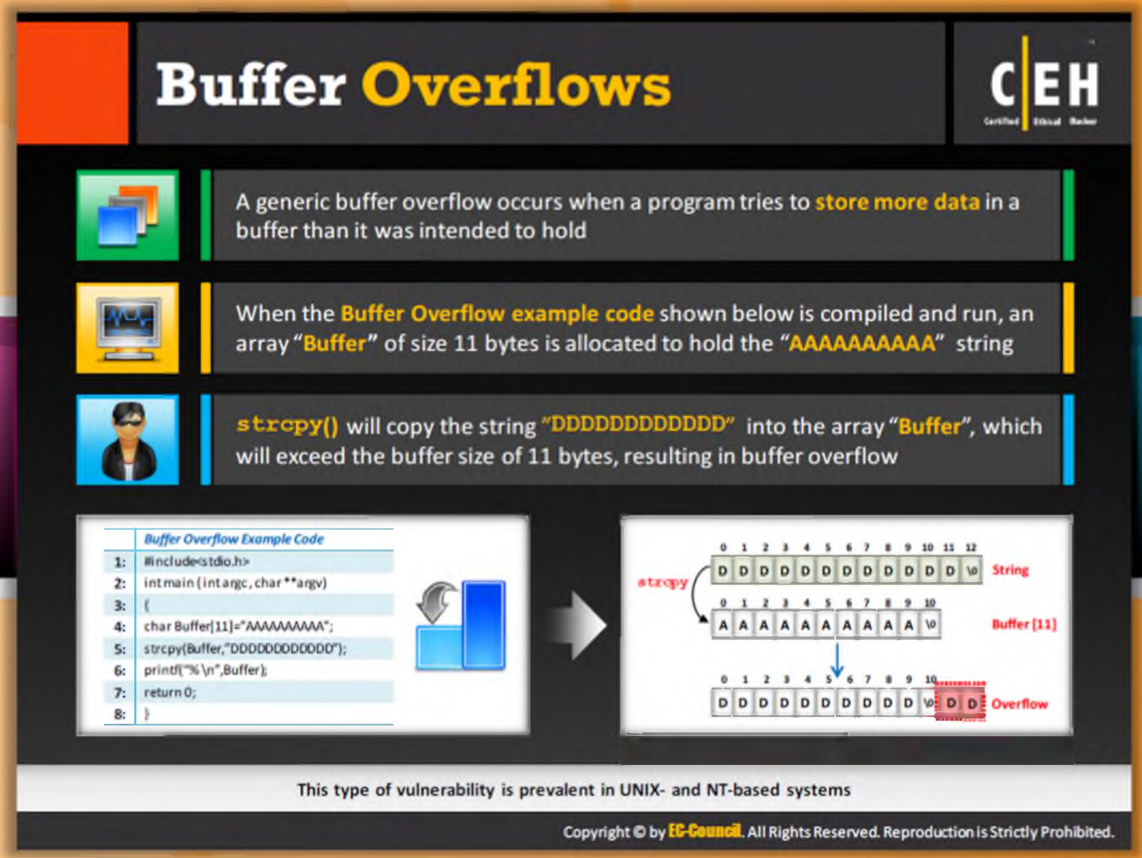
Many applications and programs are vulnerable to buffer overflow attacks. This is often overlooked by application developers or programmers. Though it seems to be simple, it may lead to severe consequences. To avoid the complexity of the buffer overflow vulnerability subject, we have divided it into various sections. Before going technically deep into the subject, first we will discuss buffer overflow concepts.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

This section describes buffer overflows, various kinds of buffer overflows (stack-based and heap-based), stack operations, shellcode, and NOPs.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Buffer Overflows

Buffers have **data storage capacity**. If the data count exceeds the original, a buffer overflow occurs. Buffers are developed to maintain finite data; additional information can be directed wherever it is needed. The extra information may **overflow** into neighboring buffers, **destroying** or overwriting the **legal data**. For example, the following C program illustrates how a buffer overflow attack works, where an attacker easily **manipulates** the code:

```
#include<stdio.h>
int main (int argc , char **argv)
      {
            char target[5]="TTTT";
            char attacker[11]="AAAAAAAAAA";
            strcpy( attacker," DDDDDDDDDDDDDD");
            printf("% \n",target);
            return 0;
      }
```


Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

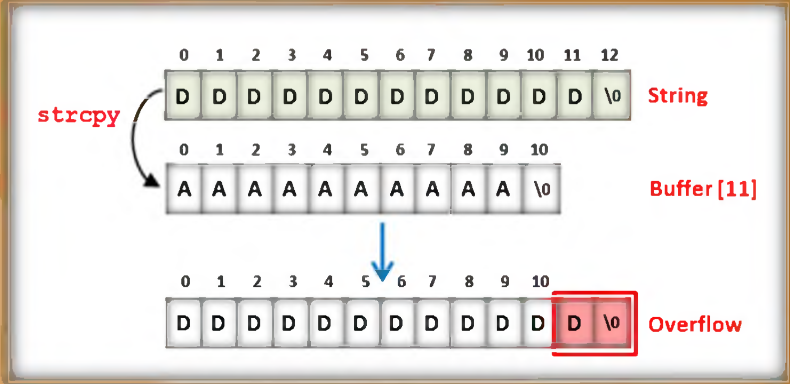


FIGURE 18.1: Buffer Overflows

The program seems to be just another normal program written by a programmer. However, the crux of this code lies in a small **manipulation** by the attacker, if examined closely. The actual problem is explained step-by-step as follows:

1. During compilation of the program, the following lines of code are executed:

```
char target[5]="TTTT";
char attacker[11]="AAAAAAAAAA";
```

- At this point, a buffer called "target," that can hold up to 5 characters, is created
- Then, the program places 4 Ts into the "**target**" buffer
- The program then creates a buffer called "attacker" that can hold up to 11 characters
- Then, the program places 10 As into the "attacker" buffer
- The program compiles these two lines of code

The following is a snapshot of the memory in the system. The contents of the target and **attacker buffer** are placed in the memory along with null characters, \0.

```
\0 T T T T
\0 A A A A
 A  A  A A A A
```

Stack Memory initially

1. After compiling the previously mentioned two lines of code, the compiler compiles the following lines of code:

```
strcpy( attacker," DDDDDDDDDDDDD");
printf("% \n",target);
```

- Here, in this line of code, the **sting copy function** is used, which copies the 13 characters of the letter D into the attacker buffer

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

- After this, the program prints the content of the **target buffer**

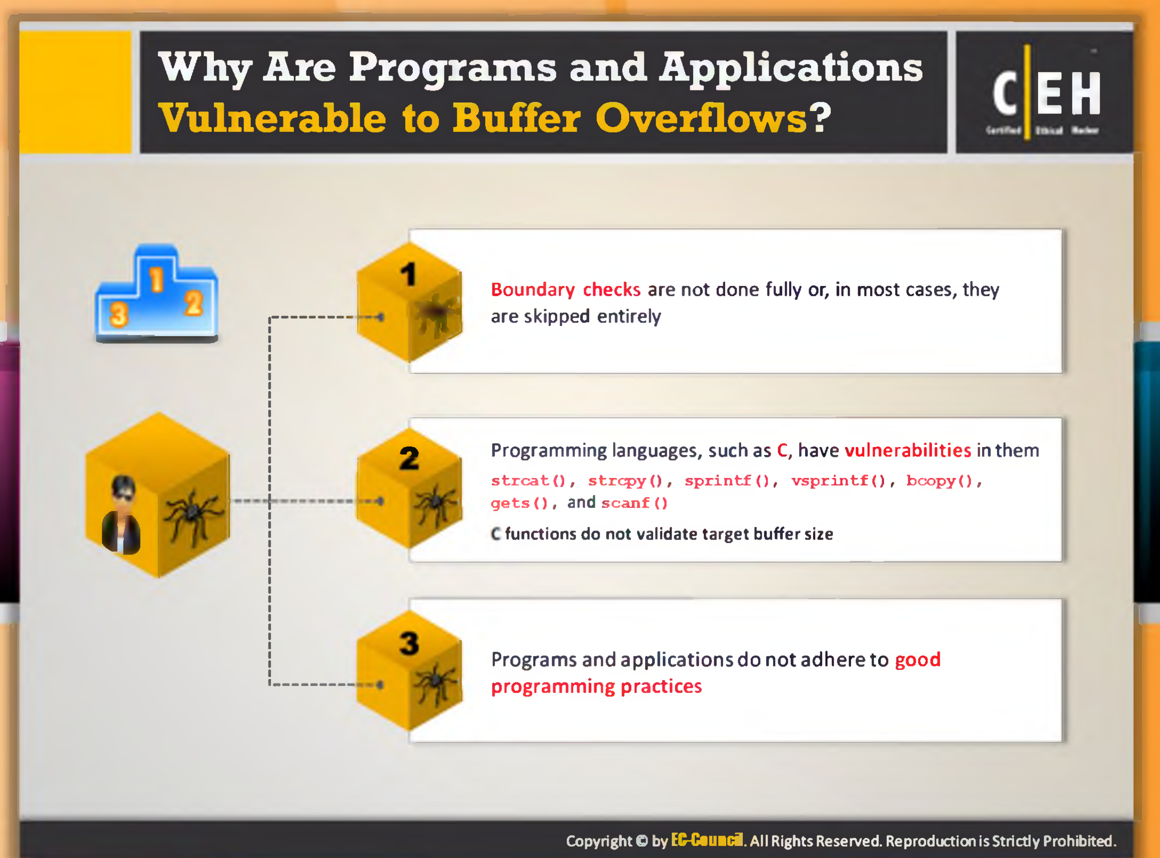
2. The strncpy function of the C program copies the 13 D characters into the attacker buffer, whose memory space is only 11 characters. Because there is no space for the remaining "D" characters, it eats up the memory of the "target" buffer, **destroying** the contents of the "target" buffer. Here is the snapshot of the system memory after the strncpy function is executed:

```
\0\0 D D  D
D  D  D D  D
D  D  D  D  D
```

This is how buffer overflow occurs:

A program, which seemed to be less problematic, created a **buffer overflow attack** just by manipulating one command. In the current scenario, the focus is primarily on the **Application Programming Interface** (API), which is a set of programming **conventions facilitating** direct communication with another piece of code; and the protocol, which is a set of data and commands to be passed between programs. It is a fact that many programs use a standard code set provided by the operating system when they want to use a protocol. The **APIs** associated with a program and the concerned **protocol** determines the nature of information that can be **exchanged** by the program. For instance, consider a simple login form. The login program can define the length of the input that can be accepted as the user name. However, if the program does not check for length, it is possible that the **storage space** allotted for the data may be used up, causing other areas in the memory to be used. If an attacker is able to detect this vulnerability, he or she can execute arbitrary code by causing the web application to act **erroneously**.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

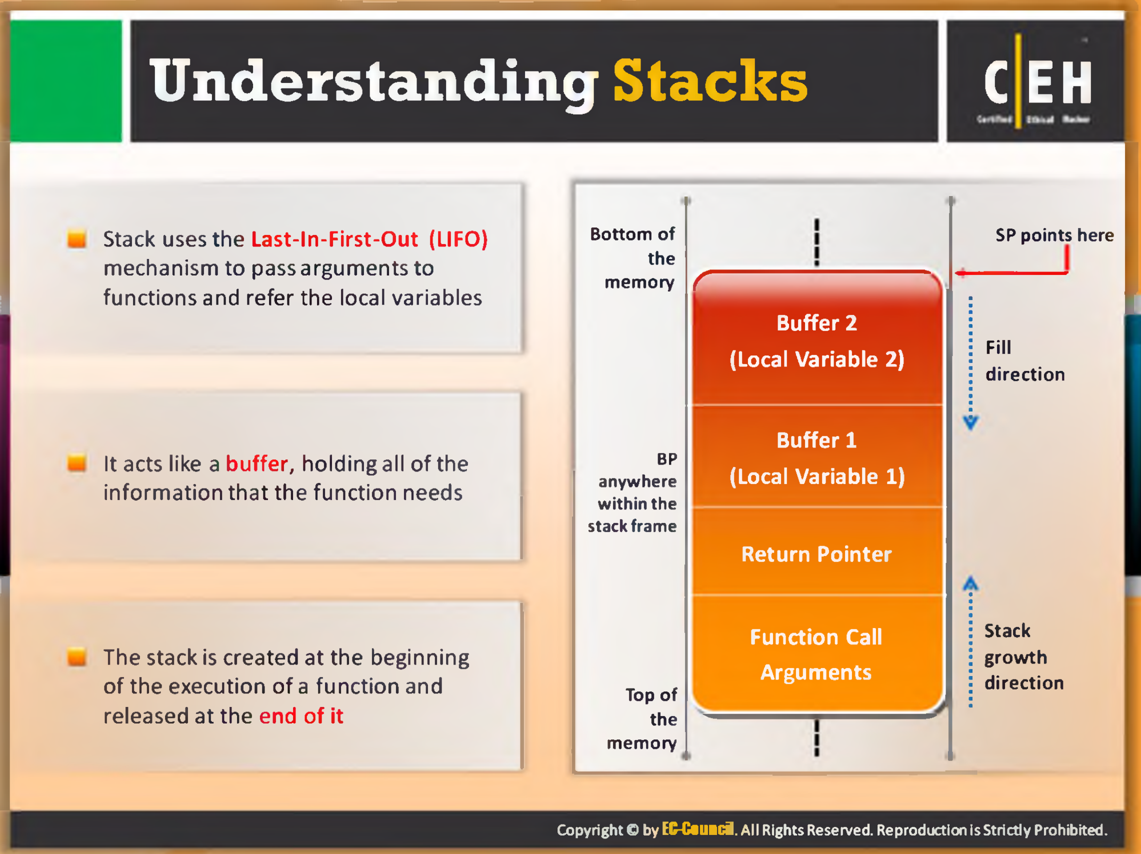


Why Are Programs and Applications Vulnerable to Buffer Overflows?

In a completely **networked world**, no organization can afford to have its server go down, even for a minute. In spite of organizations taking **precautionary measures**, **exploits** are finding their way in to **disrupt** the networks due to the following reasons:

- Pressure on the deliverables—programmers are bound to make mistakes, which are overlooked most of the time
- **Boundary checking** is not done or it is skipped in many of the cases
- Programming languages (such as C) that programmers still use to develop **packages** or applications contain errors
- The strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets(), and scanf() calls in the C language can be abused because the functions quit **testing** if any buffer in the stack is not as large as **data copied** into that buffer
- **Good programming practices** are not followed

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Understanding Stacks

A stack is a contiguous block of memory containing data. A close look at how memory is structured is shown as follows:

Code Segment

When a program runs, both code and data are loaded into memory. The code refers to the area where the instructions for the program are located. This segment contains all the compiled executable code for the program. Write permission to this segment is disabled here, as the code by itself does not contain any variables, and therefore has no need to write over itself. By having the read-only and execute attributes, the code can be shared between different copies of the program that are executing simultaneously.

Data Segment

The next section refers to the data, initialized and/or un-initialized, required by the running of the code instructions. This segment contains all the global data for the program. A read-write attribute is given, as programs would change the global variables. There is no 'execute' attribute set, as global variables are not usually meant for execution.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Stack Segment

Consider the stack as a **single-ended data structure** with first in, last out data ordering. This means that when two or more objects/elements are "**pushed**" onto the stack, to retrieve the first element, the subsequent ones have to be "**popped**" off of the stack. In other words, the most recent element remains on top of the stack. As shown previously, there is a progression from a **lower memory address** to a **higher memory address** as one moves down the stack.

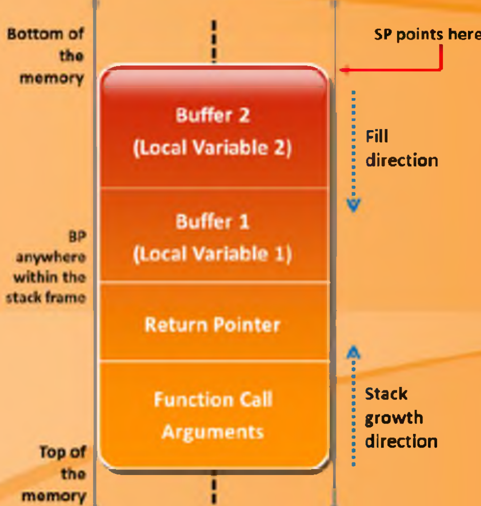
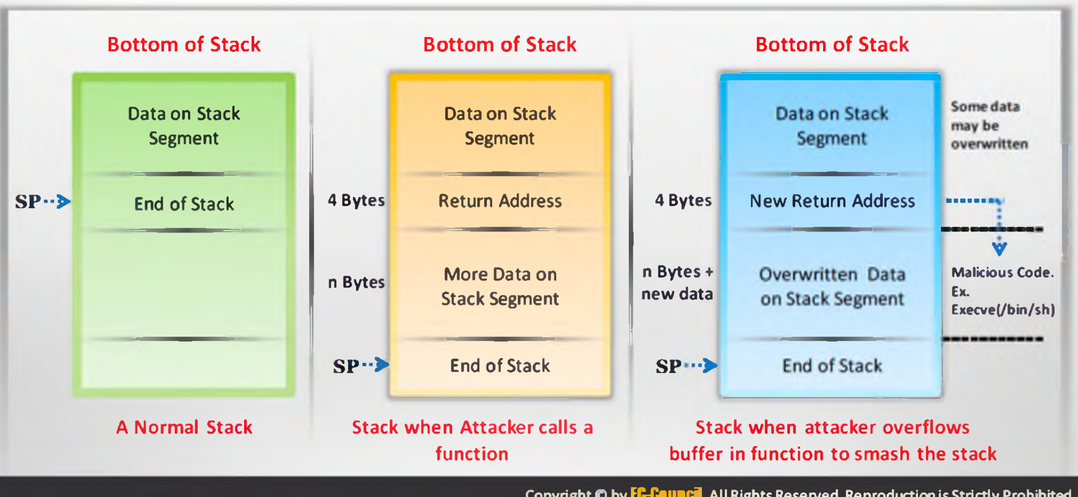


FIGURE 18.2: Stack Segment

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

Stack-Based Buffer Overflow

- A stack-based buffer overflow occurs when a buffer has been **overrun in the stack space**
- Attacker **injects malicious code** on the stack and overflows the stack to overwrite the return pointer so that the flow of control switches to the malicious code



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Stack-based Buffer Overflow

Stack-based buffer overflows have been considered the common type of exploitable programming errors found in software applications. A stack overflow occurs when data is written past a buffer in the stack space, causing unpredictability that can often lead to compromise.

Since in the eyes of the non-security community, stack overflows have been the prime focus of security vulnerability education, these bugs are becoming less prevalent in mainstream software. Nevertheless, they are still important and warrant further examination and ongoing awareness.

Over 100 functions within LibC have security implications. These implications vary from as little as "pseudo randomness not sufficiently pseudorandom" (for example, srand ()) to "may yield remote administrative privileges to a remote attacker if the function is implemented incorrectly" (for example, printf ()).

The overflow can overwrite the return pointer so that the flow of control switches to the **malicious code**. C language and its **derivatives** offer many ways to put more data than **anticipated** into a buffer.

Consider an example given as follows for simple **uncontrolled** overflow:

- The program calls the bof () function

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

- Once in the bof () function, a string of 20 As is copied into a buffer that holds 8 bytes, resulting in a buffer overflow

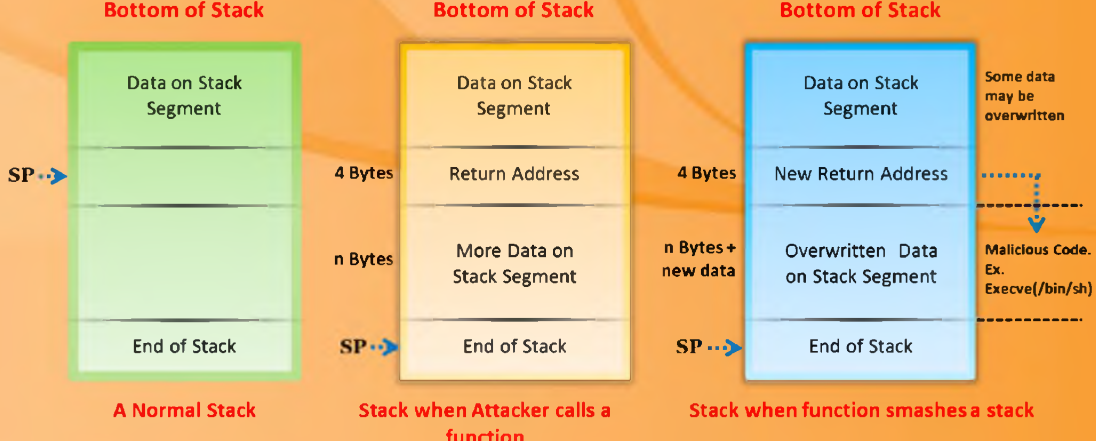


FIGURE 18.3: Stack-based Buffer Overflow

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Understanding Heap

CEH

- Heap is a memory segment used by a program and is allocated dynamically at run time with functions such as `malloc()`, `calloc()`, `realloc()` in C and using `new` operator in C++
- Control data is stored on the heap along with the data allocated using the malloc interface
- Heap stores all instances or attributes, constructors, and methods of a class or object

A= malloc(10);
C= malloc(4);
AAAAAAA AAA
BBBBB BBB
CC CC
Memory Contents
Memory Contents
Memory Contents
Control Data
Control Data
Control Data

Simple Heap Contents

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Understanding Heap

The heap is an area of memory utilized by an application and allocated dynamically at runtime. It is common for a buffer overflow to occur in the heap memory space, and exploitation of these bugs is different from **stack-based buffer overflows**. Heap overflows have been the prominent software security bugs. Unlike stack overflows, heap overflows can be inconsistent and can have varying exploitation techniques and consequences.

Heap memory is different from stack memory; in that heap, memory is persistent between functions, with memory allocated in one function remaining allocated until explicitly freed. This means that a heap overflow can occur, but it is not noticed until that section of memory is used later. There is no concept of saved EIP in relation to a heap, but other important things are stored in the heap and can be broken by overflowing dynamic buffers.

From a primitive point of view, the heap consists of many blocks of memory, some of which are allocated to the program and some are free, but allocated blocks are often placed in adjacent places of memory.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

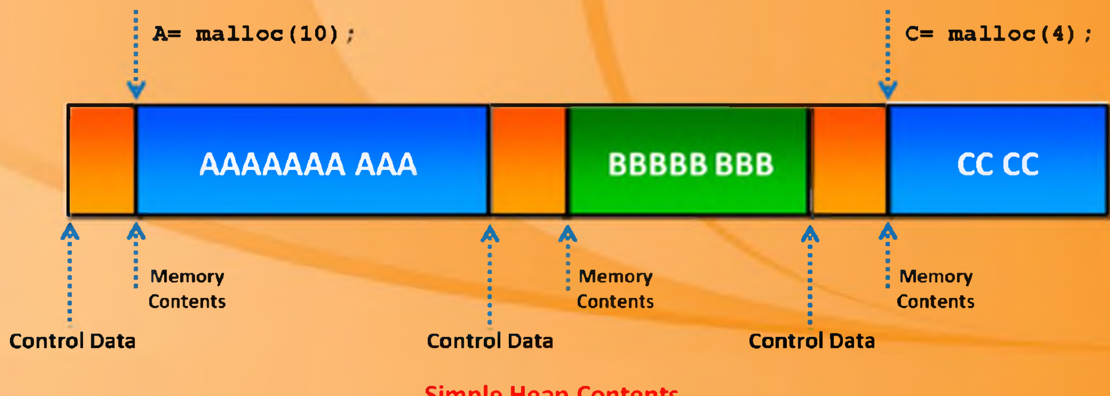


FIGURE 18.4: Understanding Heap

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

Heap-Based Buffer Overflow

CEH Certified Ethical Hacker

- If an application copies the data without checking whether it fits into the target destination, attackers can supply the application with a large data, overwriting the heap management information
- Attackers overflow buffers on the lower lower part of heap, overwriting other dynamic variables, which can have unexpected and unwanted effects

input=malloc(20); output=malloc(20);

XXXXXXXXXXXXXXXXXXXX | "normal output\0"

Heap: Before Overflow

input=malloc(20); output=malloc(20);

fnordfnordfnordfnordf | fno | rdfnordfnord\0

Heap: After Overflow

Note: In most environments, this may allow the attacker to control the program's execution

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Heap-based Buffer Overflow

The heap is an **area of memory utilized** by an application and allocated dynamically at runtime. It is common for buffer overflows to occur in the heap memory space, and exploitation of these bugs is different from that of stack-based buffer overflows. Heap overflows have been the prominent discovered software security bugs. Unlike stack overflows, heap overflows can be inconsistent and have varying exploitation techniques.

An application dynamically allocates heap memory as needed. This allocation occurs through the function call malloc (). The malloc () function is called with an argument specifying the number of bytes to be allocated and returns a pointer to the allocated memory.

- Variables that are **dynamically allocated** with functions, such as malloc (), are created on the heap.
- An attacker overflows a buffer that is placed on the lower part of heap, overwriting other dynamic variables, which can have **unexpected** and unwanted effects.
- If an application copies data without first **checking** whether it fits into the target destination, the attacker could supply the application with a piece of data that is large, overwriting **heap management** information.
- In most environments, this may allow the attacker to **control** over the program's execution.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Check what happens to the program when input grows past the allocated space. This happens because there is no control over its size. Run the program several times with different input strings.

```
[root@localhost]# ./heap1 hackshacksuselessdata
input at 0x8049728: hackshacksuselessdata
output at 0x8049740: normal output
normal output
[root@localhost]# ./heap1
hacks1hacks2hacks3hacks4hacks5hacks6hacks7hackshackshackshackshack
shacks
input at 0x8049728:
hacks1hacks2hacks3hacks4hacks5hacks6hacks7hackshackshackshackshack
shacks
output at 0x8049740: hackshackshackshacks5hacks6hacks7
hackshacks5hackshacks6hackshacks7
[root@localhost]#./heap1"hackshacks1hackshacks2hackshacks3hackshacks4wh
at have I done?"
Input at 0x8049728: hackshacks1hackshacks2hackshacks3hackshacks4 what
have I done?
output at 0x8049740: what have I done?
what have I done?
[root@localhost]#
```

Thus, overwriting variables on the heap is easy and does not always cause crashes.

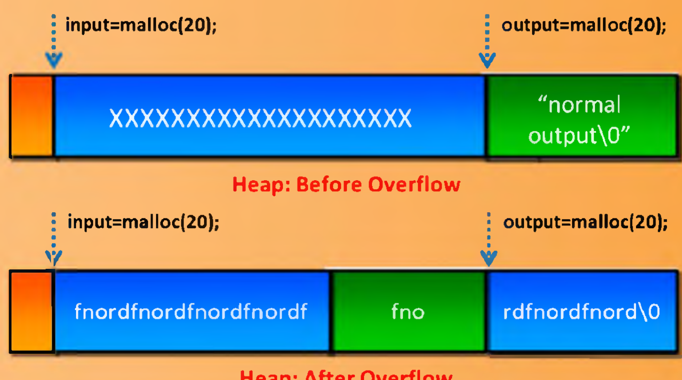
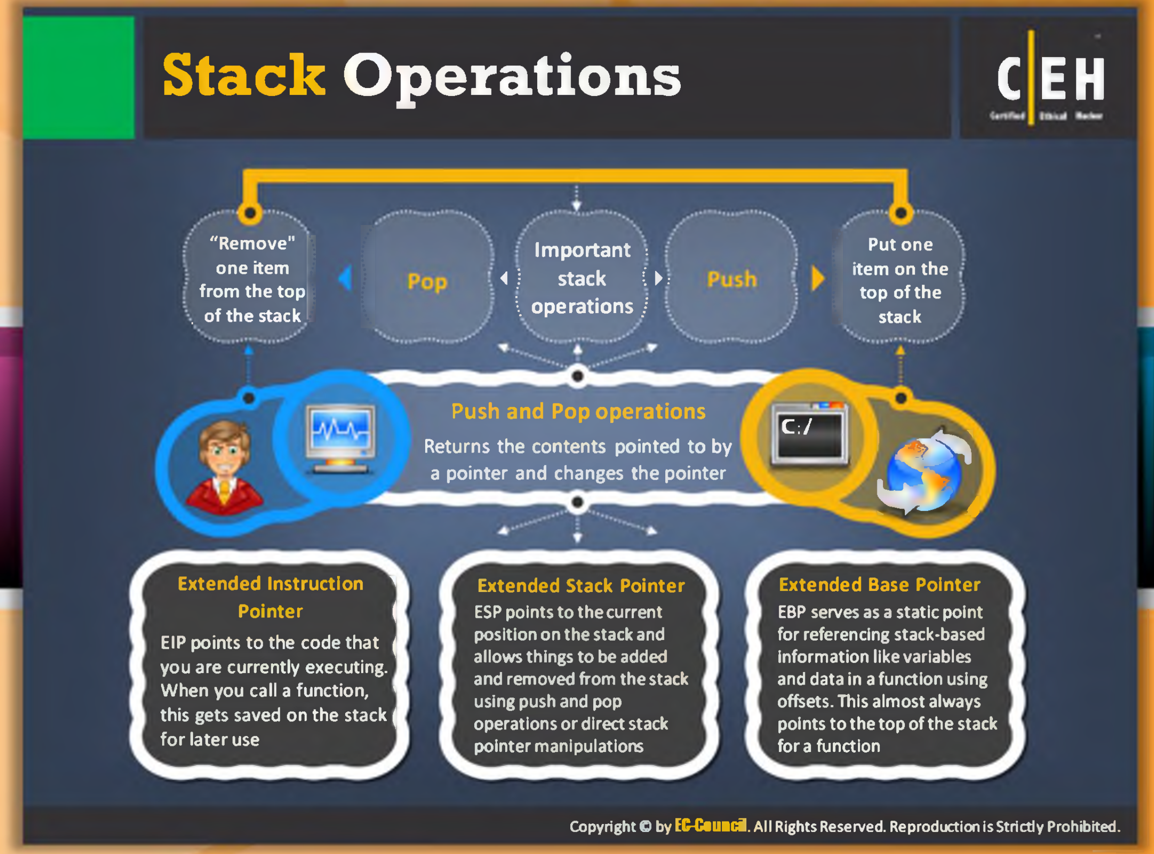


FIGURE 18.5: Heap-based Buffer Overflow

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Stack Operations

A stack is implemented by the system for programs that run on the system. A variable can be deployed within the processor itself and memory can also be allocated. The variable is called the "**register**" and the region of memory is the "**stack**." The register used to refer to the stack as the "**Stack Pointer**" or SP. The SP points to the top of the stack, while the bottom of the stack is a fixed address. The kernel adjusts the stack size **dynamically** at run time.

A stack frame, or record, is an activation record that is stored on the stack. The stack frame has the following: the parameters to a function, its local variables, and the data required to restore the previous stack frame, along with the value of the instruction pointer (pointer that points the next instruction to be **fetched** at the function call) at the time of the function call.

The majority functionality of the stack involves adding and removing items from the stack. This is **accomplished** with the help of two major operations. They are **push** and **pop**.

When the program is loaded, the stack pointer is set to the highest address. This will be the topmost item in the stack. When an item is pushed onto the stack, two events take place. **Subtracting** the size of the item in bytes from the initial value of the **pointer** reduces the stack pointer. Next, all the **bytes** of the items in consideration are copied into the region of the stack segment to which the stack pointer now points.

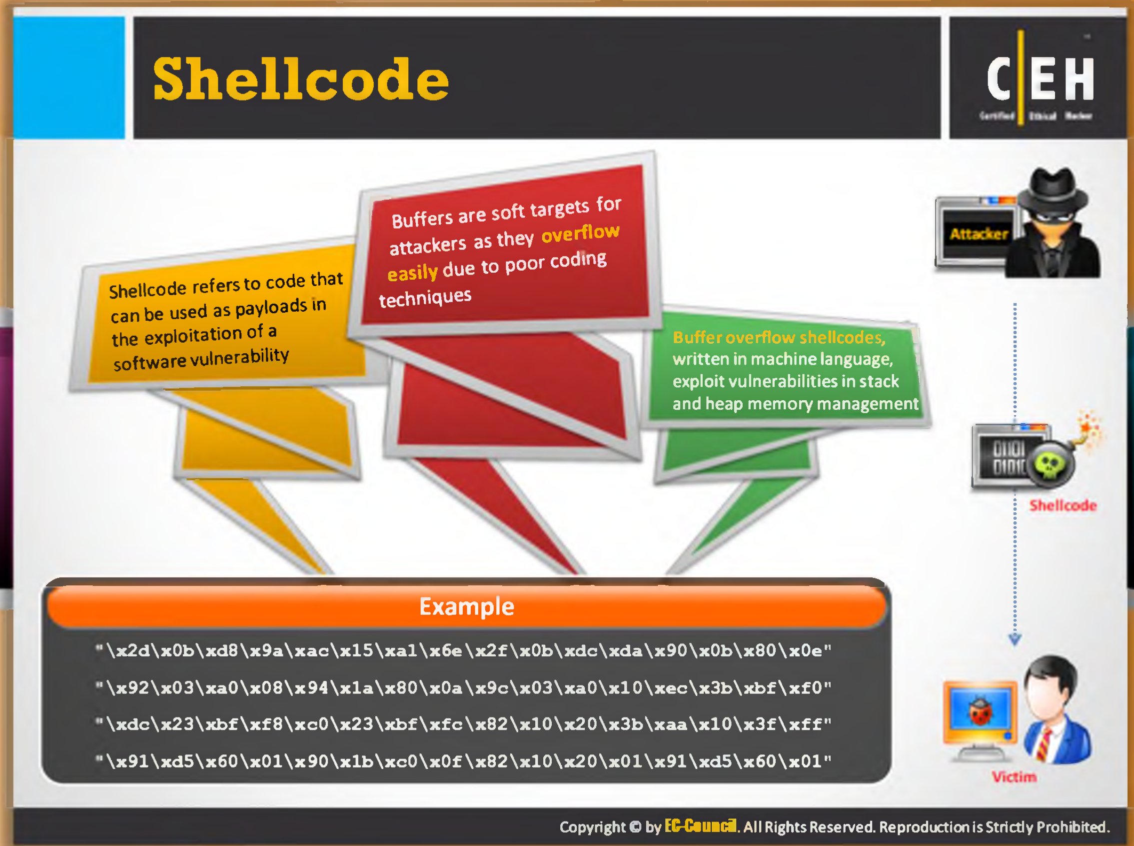
Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Similarly, when an item is popped from the stack, the size of the item in **bytes** is added to the stack pointer. However, the copy of the item continues to reside on the stack. This will eventually be overwritten when the next push operation takes place. Based on the **stack design** implementation, the **stack** can come down (toward lower memory addresses) or go up (toward higher memory addresses).

When a procedure is called, it is not the only item that pushes onto the stack. Among others is the address of the calling procedure's instruction immediately following the procedure call. This is followed by the **parameters** to the called function. As the called function completes, it would have **popped** its own local variables off the stack. The last **instruction** the called function runs is a special instruction called a return. The top values of the stack are popped up and loaded into the IP by the assembly language, a **special processor** instruction. At this point, the stack will have the address of the next instruction of the calling procedure in it. The other concept that the reader needs to **appreciate** in order to understand the complete essence of **stack overflows** is the **frame pointer**.

Apart from the stack pointer, which points to the top of the stack, there is a **frame pointer (FP)** that points to a **fixed location** within a frame. Local variables are usually referenced by their offsets from the stack pointer. However, as the stack operations take place, the value of these offsets vary. Moreover, on processors such as **Intel-based processors**, accessing a variable at a known distance from the stack pointer requires multiple instructions. Therefore, a second register may be used for **referencing** those variables and parameters whose relative distance from the frame pointer does not change with stack operations. On Intel processors, the **base pointer** (BP), also known as the **Extended Base Pointer** (EBP), is used.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Shellcode

Shellcode is a small code used as **payload** in the exploitation of a software vulnerability. Shellcode is a technique used to **exploit stack-based overflows**. Shellcodes **exploit** programming bugs in **stack handling**. Buffers are soft targets for attackers as they overflow easily if the conditions match. Buffer overflow shellcodes, written in **assemble language**, exploit vulnerabilities in stack and **heap memory management**

For example, the VRFY command helps the attacker to identify potential users on the target system by verifying their email addresses. In addition, sendmail uses a set user ID of root and runs with **root privileges**. If the attacker connects to the sendmail daemon and sends a block of data consisting of **1,000 a's** to the **VRFY command**, the VRFY buffer is overrun as it was only designed to hold **128 bytes**.

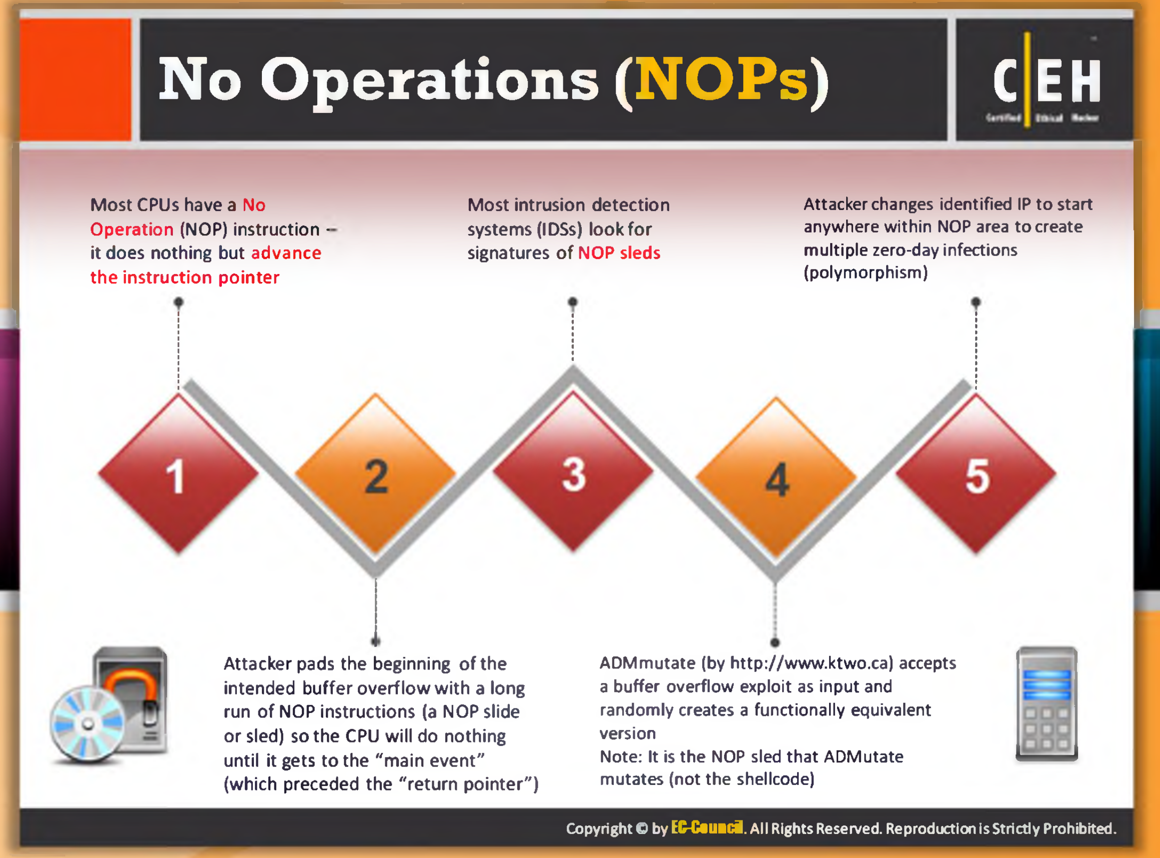
However, instead of sending 1000 a's, the attacker can send a specific code that can overflow the buffer and execute the command /bin/sh. In this case, when the attack is carried out, a special assembly code "**egg**" is transferred to the VRFY command, which is a part of the actual string used to overflow the buffer. When the VRFY buffer is overrun, instead of the **offending** function returning to its original **memory address**, the attacker executes the **malevolent machine code** that was sent as a part of the buffer overflow data, which executes /bin/sh with root privileges.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

The following illustrates what an egg, specific to Linux X86, looks like:

```
Char shellcode [ ] =
"\xeb\xlf\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```


Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



No Operations (NOPs)

Even the best guess may not be good enough for an attacker to find the **right address** on the stack. If the attacker is off by one byte, more or less, there can be a segmentation violation or an **invalid instruction**. This can even cause the system to crash. The attacker can increase the odds of finding the right address by **padding** his or her code with **NOP instructions**.

A NOP is just a command telling the **processor** to do nothing other than take up time to process the **NOP instruction** itself. Almost all processors have a NOP instruction that performs a null operation. In the Intel architecture, the NOP instruction is **1 byte** long and translates to **0x90** in **machine code.** A long run of NOP instructions is called a **NOP slide** or sled, and the CPU does nothing until it gets back to the main event (which precedes the "return pointer").

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

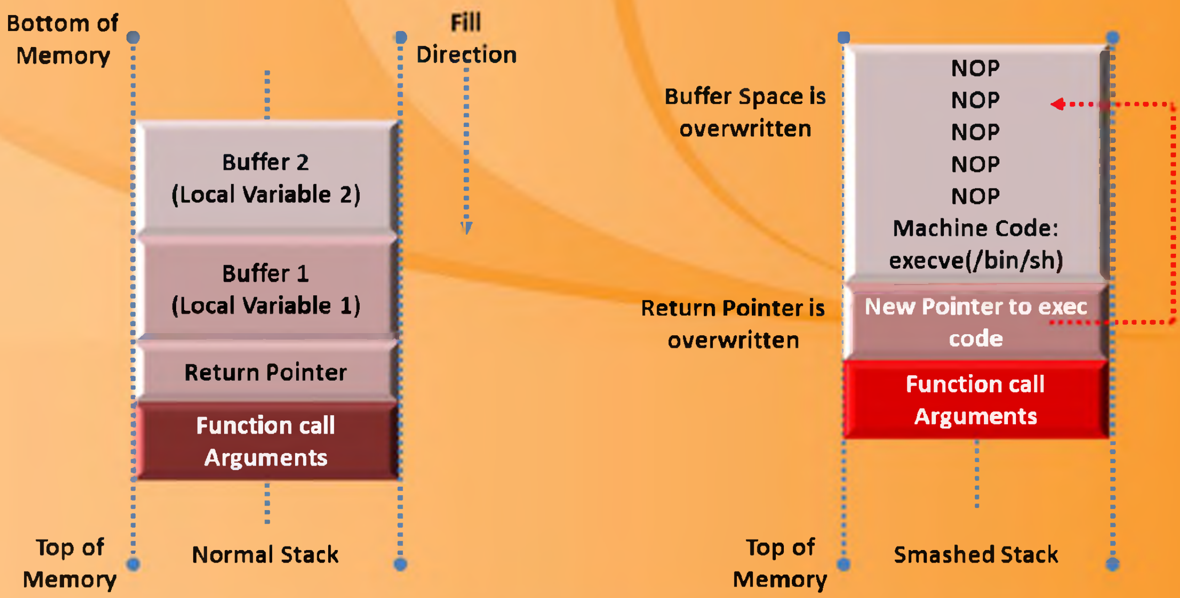
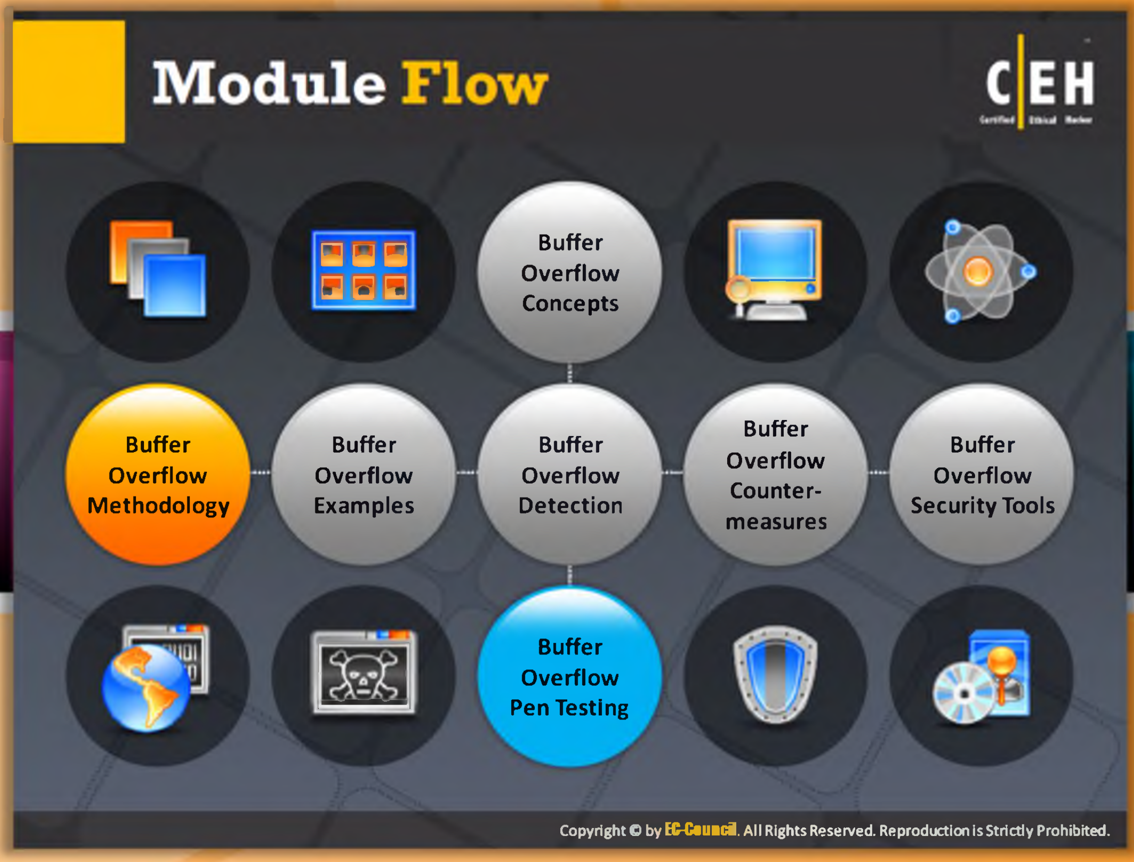


FIGURE 18.6: No Operations (NOPs)

By including NOPs in advance of the **executable code**, the attacker can avert a **segmentation violation** if an overwritten return pointer lands execution in the NOPs. The program can continue to execute down the stack until it gets to the attacker's exploit. In the **preceding illustration**, the attacker's data is written into the allocated buffer by the function. As the data size is not checked, the return pointer can be overwritten by the attacker's input. With this method, the attacker places the **exploited machine's code** in the buffer and overwrites the return pointer so that when the function returns, the attacker's code is executed.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



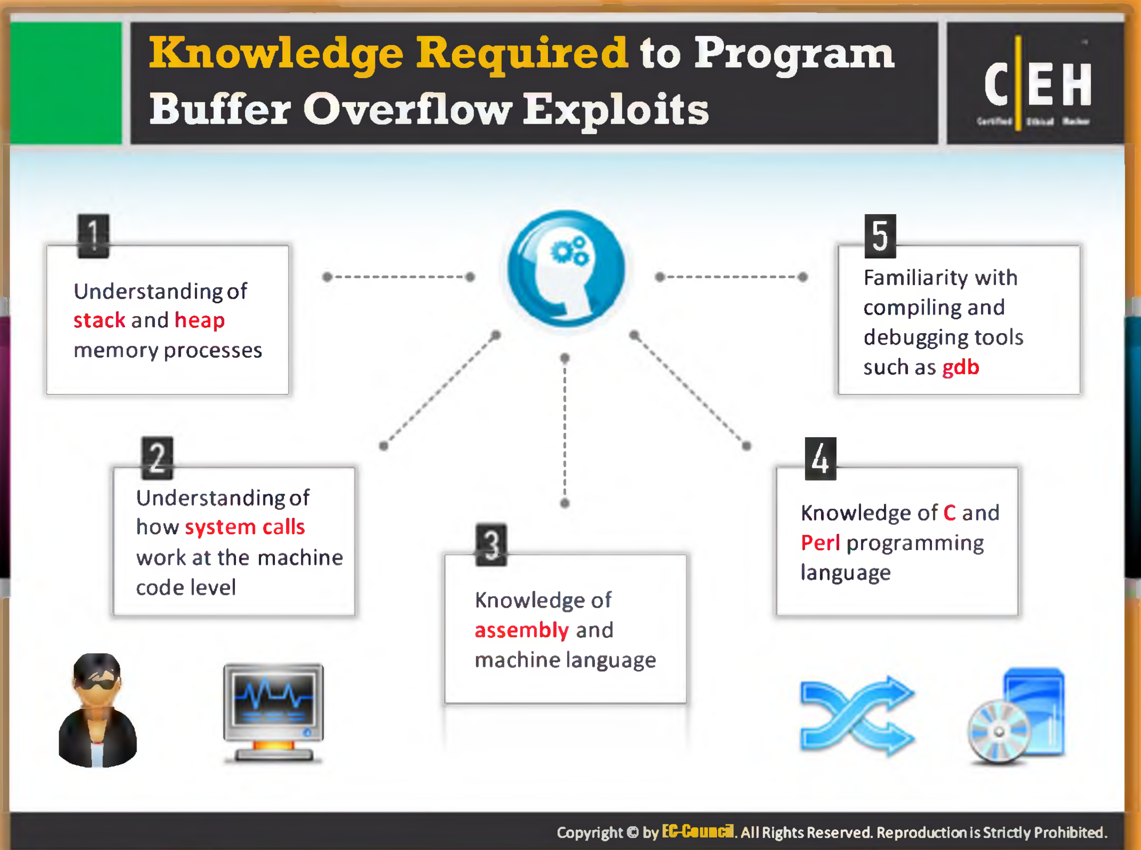
Module Flow

So far, we have discussed the basic buffer overflow concepts, Now we will discuss the buffer overflow methodology.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

This section describes requirements to program buffer overflow exploits, buffer overflow steps, and buffer overflow vulnerabilities.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Knowledge Required to Program Buffer Overflow Exploits

Logically, the question that arises is why are stacks used when they pose such a **threat to security**? The answer lies in the **high-level**, **object-oriented programming languages**, where procedures or functions form the basis of every program.

A stack is used for **storing context**. For instance, if a procedure simply pushes all its local variables onto the **stack** when it enters, and pops those off when it is over, its entire context is cleaned up so that when the procedure calls another procedure, the called procedure can do the same with its context, without the aid of the **calling procedure's** data. The flow of control is determined by the procedure or function, which is resumed after the current one is done. The stack implements the **high-level abstraction**. Apart from this, the stack also serves to **dynamically allocate** local variables used in functions, passing parameters to functions, and to return values from the function.

In fact, though several applications are written in C, programs written in C are particularly susceptible to **buffer overflow attacks**. This is due to the fact that direct pointer variations are permitted in C. Direct, **low-level memory access** and the **pointer arithmetic** is provided by C without checking the bounds. Moreover, the standard C library provides unsafe functions (such

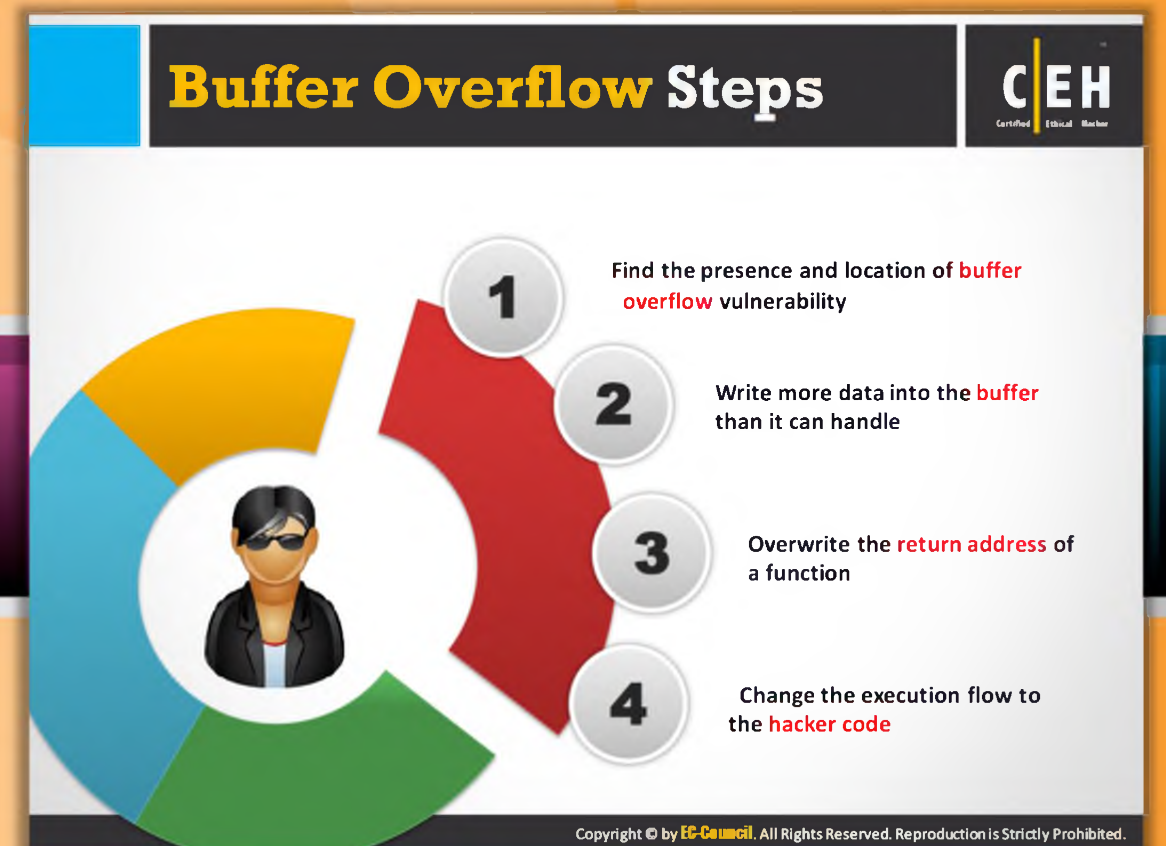
Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

as gets) that write an **unbounded** amount of user's input into a **fixed size buffer** without any **boundary checking**.

To program buffer overflow exploits, you should be **acquainted** with the following aspects:

- Understanding of stack and heap memory processes
- Understanding of how **system calls** work at the machine code level
- Knowledge of assembly and machine language
- Knowledge of C and Perl programming language
- **Familiarity with compiling and debugging tools** such as gdb
- exec( ) system calls How to guess some key parameters
- How to guess some key parameters

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Buffer Overflow Steps

Buffer overflow can be carried out in four steps:

Step 1: In order to perform a **buffer overflow attack**, first you should check whether the **target** application or program is **vulnerable** to buffer overflow or not. Typically buffer overflow occurs when the input entered **exceeds** the size of the buffer. If there is any potential buffer overflow vulnerability present in the program, then it displays an error when you enter a **lengthy string** (exceeding the size of buffer). Thus, you can **confirm** whether a program contains a buffer overflow vulnerability or not. If it is vulnerable, then find the location of the buffer overflow vulnerability.

Step 2: Once you find the location of the **vulnerability**, write more data into the buffer than it can handle. This causes the buffer overflow.

Step 3: When a buffer overflow occurs, it overwrites the memory. Using this advantage, you can overwrite the return address of a function with the address of the shellcode.

Step 4: When the **overwrite** occurs, the **execution flow** changes from normal to the **shell code**. Thus, you can execute anything you want.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Attacking a Real Program

C|EH Certified Ethical Hacker

Assuming that a string function is exploited, the attacker can send a long string as the input. This string overflows the buffer and causes a segmentation error

The return pointer of the function is overwritten, and the attacker succeeds in altering the flow of the execution

If the attacker inserts code as input, he or she has to know the exact address and size of the stack and make the return pointer point to the code for execution

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Attacking a Real Program

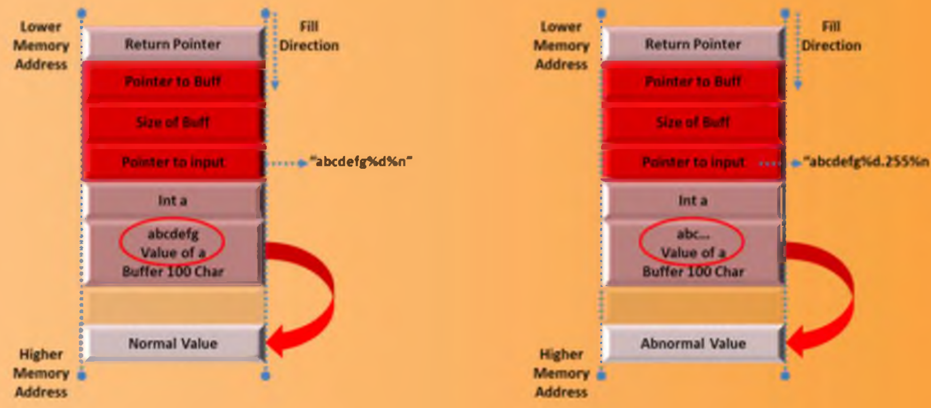


FIGURE 18.7: Attacking a Real Program

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

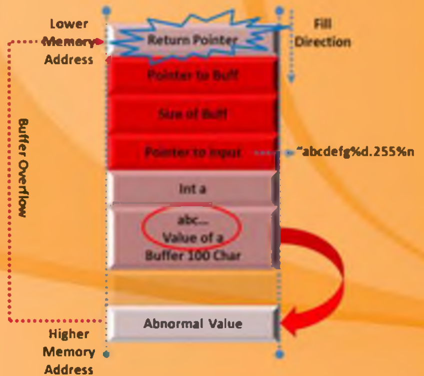


FIGURE 18.8: Attacking a Real Program

The previous illustration depicts the way an **abnormal input** causes the buffer to overflow and cause a **segmentation error**. Eventually, the return pointer is overwritten and the execution flow of the function is **interrupted**. Now, if the attacker wants to make the function **execute arbitrary code** of his or her choice, he or she can have to make the return pointer point towards this code.

When attacking a real program, an attacker has to assume that a string function is being exploited, and send a **long string** as the input. After passing the input string, the string overflows the buffer and causes a segmentation error. The return **pointer** of the function is overwritten, and the attacker succeeds in altering the **flow of execution**. If the user has to insert his or her code in the input, he or she has to:

- Know the **exact address** on the stack.
- Know the size of the stack.
- Make the **return pointer** point to his/her code for execution.

The challenges that the attacker faces are:

- Determining the **size** of the buffer.
- The attacker must know the address of the stack so that he or she can get his or her input to rewrite the return pointer. For this, he or she must **ascertain** the exact address.
- The attacker must write a program small enough that it can be passed through as input.

Usually, the goal of the attacker is to **spawn a shell** and use it to **direct further commands**.

The code to spawn a shell in C is as follows:

```
-------------------------------------------------------------------------------
#include <stdio.h>
   void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL); }
```

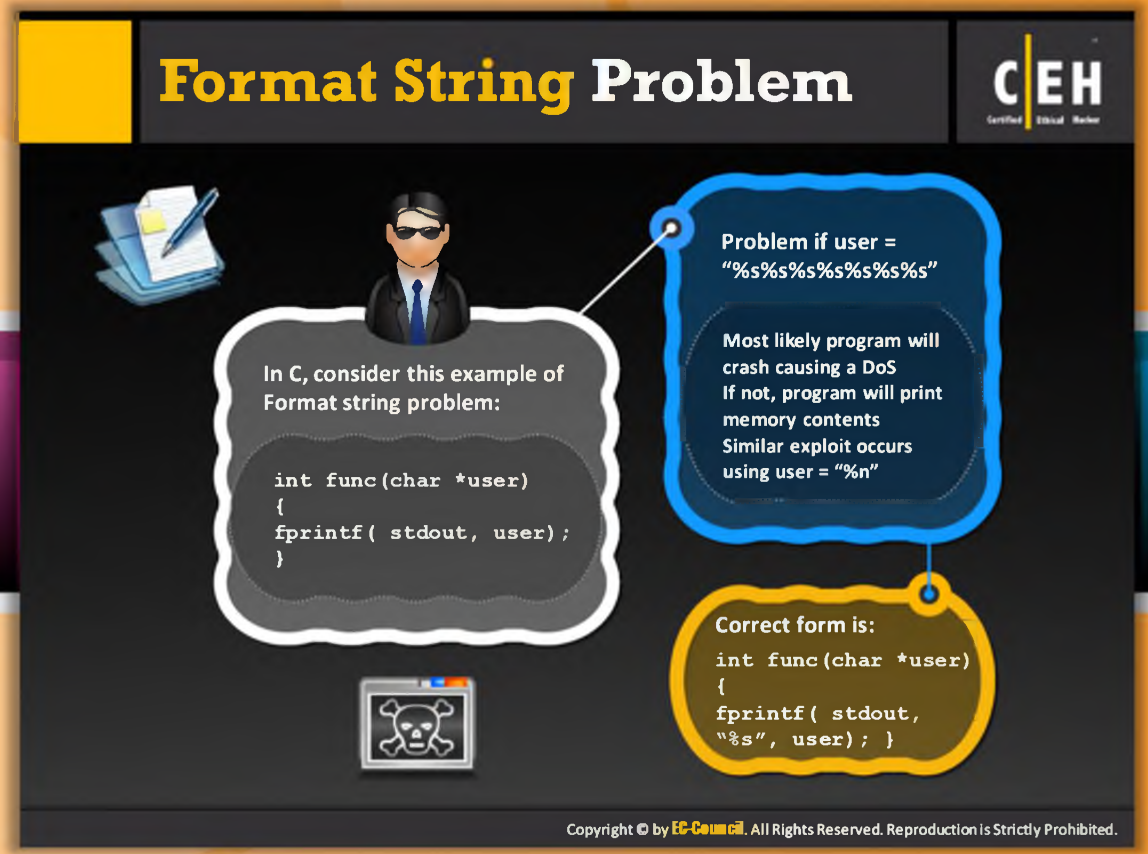

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

The attacker can place **arbitrary code** to be executed in the buffer that is to be **overflowed** and overwrite the return address so that it points back into the buffer. For this, he or she must know the exact location in the memory space of the program whose code has to be exploited. A workaround for this challenge is to use a **jump (JMP)** and a **CALL instruction**. These instructions allow **relative addressing** and permit the attacker to point to an offset relative to the instruction pointer. This **eliminates** the need to know the exact address in the memory to which the exploit code must point. As most **operating systems** mark the code pages with the read-only attribute, this makes the previously discussed workaround an **unfeasible** one. The alternative is to place the code to be executed into the stack or **data segment** and transfer control to it. One way of **achieving** this is to place the code in a **global array** in the data segment as shown in the previous code snippet.

Does the exploit work? Yes.

Nevertheless, in maximum **buffer overflow vulnerabilities**, it is the character buffer that is subjected to the attack. Therefore, any null code occurring in the shell code can be considered as the end of the string, and the **code transfer** can be terminated. The answer to this **hindrance** lies in NOP.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Format String Problem

Format string problems usually occur when the input is supplied from untrusted sources or when the data is passed as a **format string argument** to functions such as syslog(), sprintf(), etc. The format string vulnerabilities in **C/C++** can easily be exploited because of the **%n operator**. If any program contains this kind of vulnerability, then the **program's confidentiality** and the **access control** may be at risk because the format string vulnerability exploitation results in information disclosure and **execution** of **arbitrary code** without the knowledge. Thus, attackers can easily exploit the program or application containing format string **vulnerabilities**.

In C, consider this example of a format string problem:

```
int func(char *user)
{
fprintf( stdout, user);
}
```

Problem if `user = "%s%s%s%s%s%s%s"`

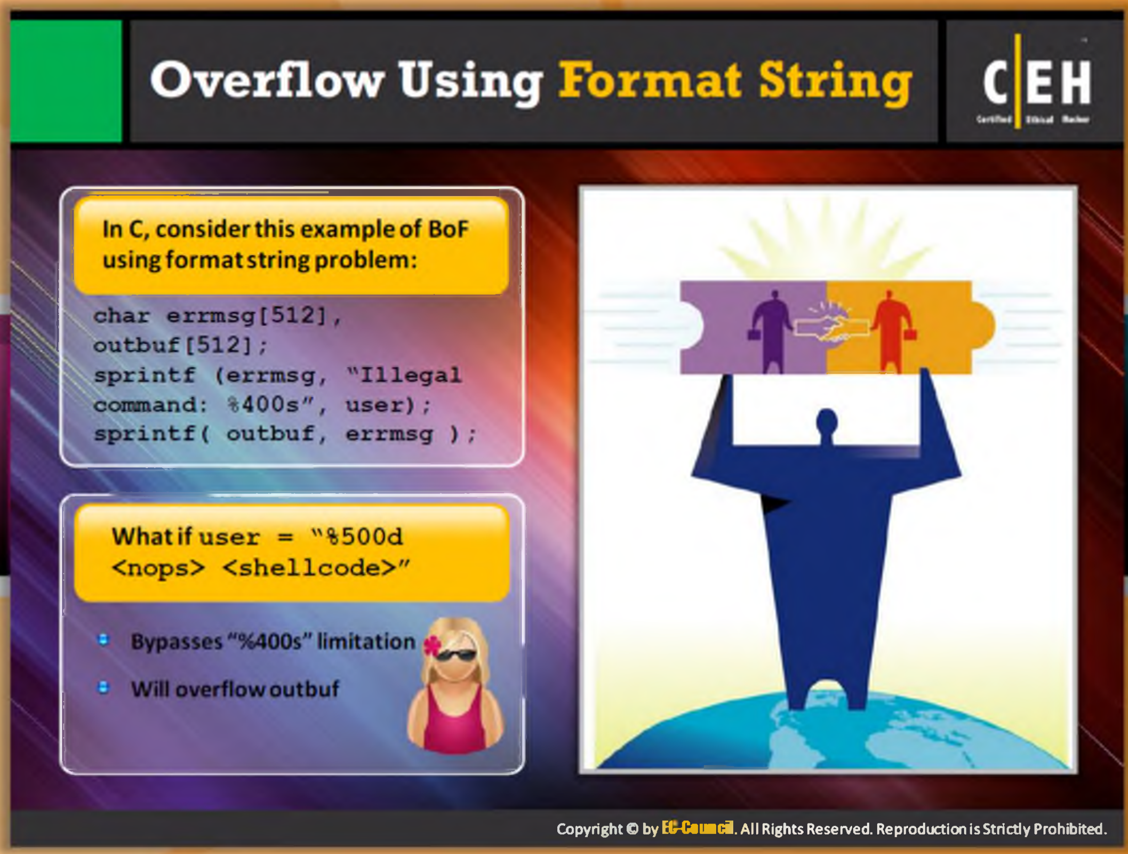
Most likely, the program will crash, causing a DoS. If not, the program will print memory contents. Full exploit occurs using user = "`%n`"

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Correct form is:

```
int func(char *user)
{
fprintf( stdout, "%s", user);
}
```


Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Overflow Using Format String

In C, the format string library functions take **variable numbers of arguments.** The format string variable is the one that is always required. The format string contains **format-directive characters** and printable characters. **Format string overflow attacks** are quite similar to that of buffer overflow attacks since in both attacks the attacker attempts to change the memory space and **consequently** runs **arbitrary code**. The only difference between these two attacks is that the attacker launches the format string overflow attack by **exploiting** the vulnerabilities in the **variadic functions**, such as format functions.

Format string overflow can be exploited in four ways:

- Memory viewing
- Updating a word present in the memory
- Making a buffer overflow by using minimum field size specifier
- Using %n **format directive** for overwriting the code

In C, consider this example of BoF using format string problem:

char errmsg[512], outbuf[512];

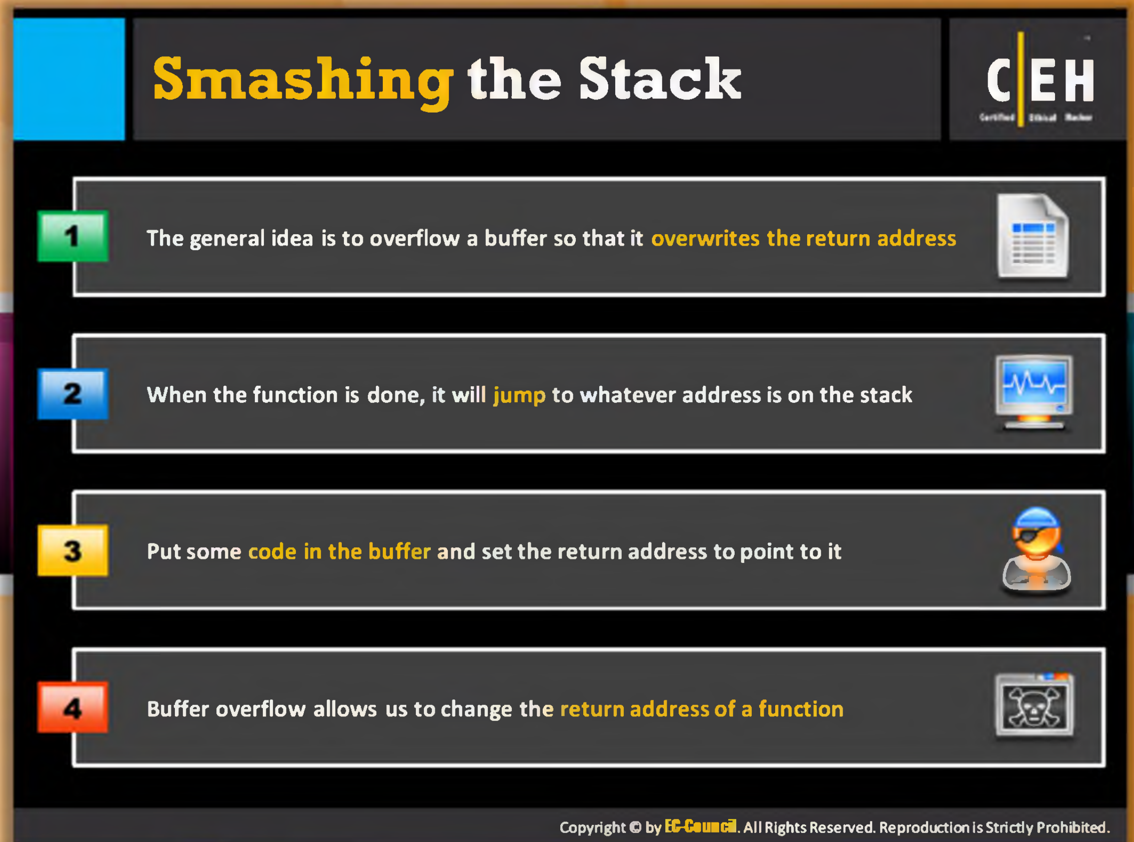
sprintf (errmsg, "Illegal command: %400s", user);

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

sprintf(outbuf, errmsg);

If user = “%500d <nops> <shellcode>”, this will bypass ““%400s” limitation and overflow outbuf. Thus, the stack smashing buffer overflow attack is carried out.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

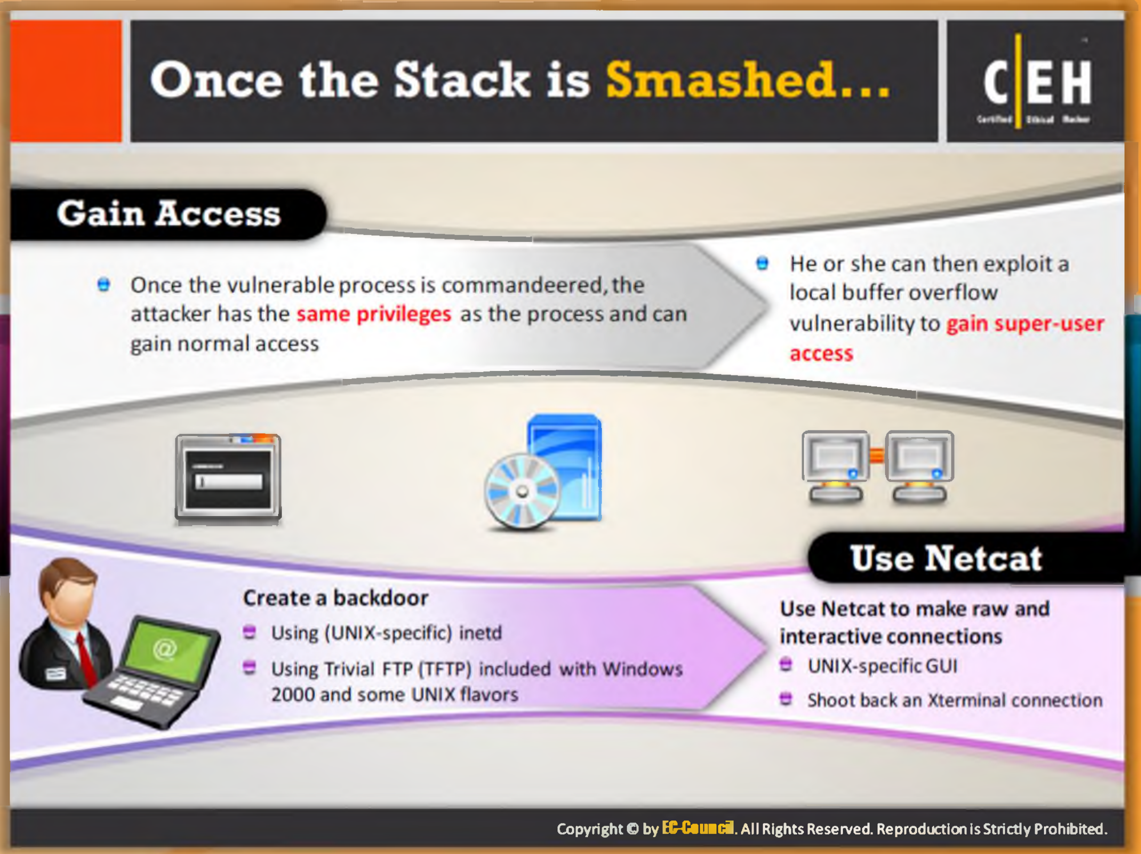


Smashing the Stack

Smashing the stack causes a **stack to overflow**. The stack is a **first-in last-out form** of buffer to hold the **intermediate results** of an operation. If you try to store more data than the stack's size, then it drops the excess data. The data that a stack holds may be critical for **system operation**.

The general idea behind **stack smashing** is to overflow a buffer which in turn overwrites the return address. If the attacker succeeds in smashing the stack, then he or she can overwrite the address on the stack with the address of **shellcode**. When the function is done, it jumps to the return address, i.e., the shellcode address. Thus an attacker can exploit the **buffer overflow vulnerability**.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



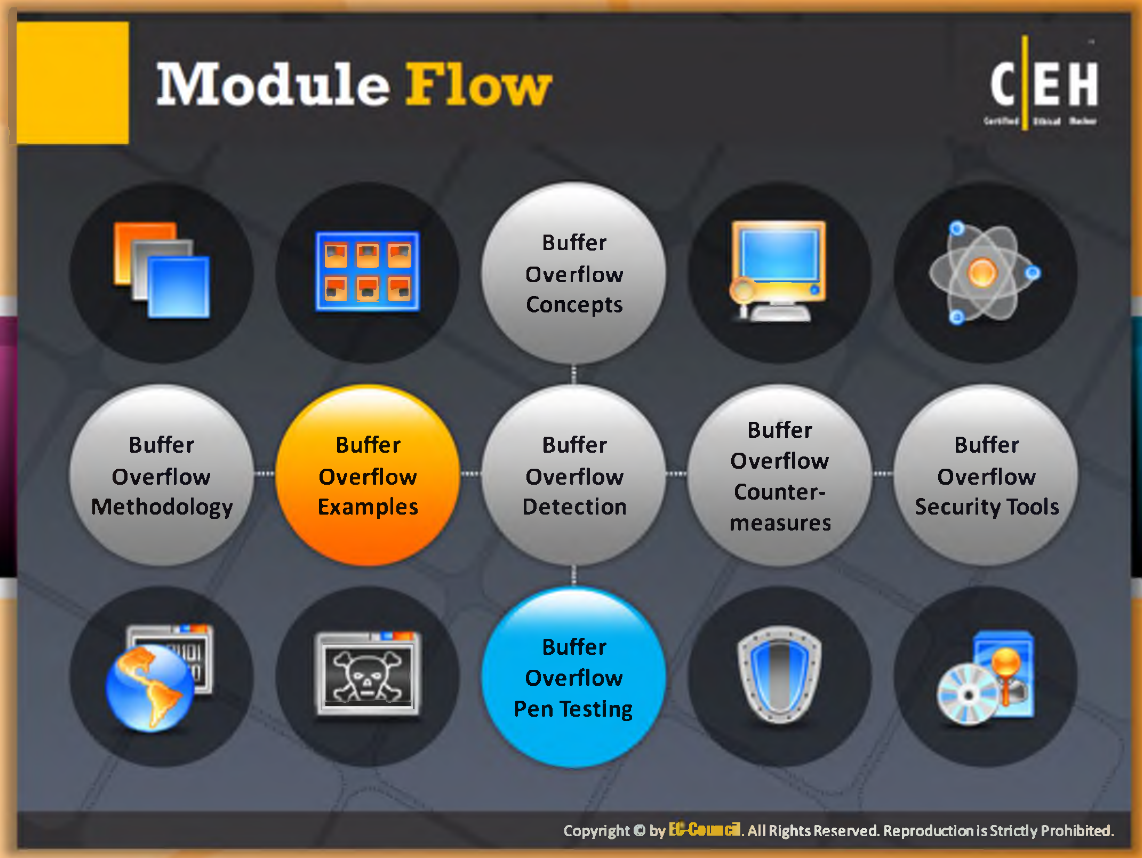
Once the Stack is Smashed

There are two parts of the attacker's input: an injection vector and a payload. They may be separate or put together. The injection vector is the correct entry-point that is tied unambiguously along with the bug itself. It is OS/target/application/protocol/encoding-dependent. On the other hand, the payload is usually not tied to bugs at all and is contained by the attacker's ingenuity. Even though it can be independent of the injection vector, it still depends on machine, processor, and so on.

Once the stack is smashed, the attacker can deploy his or her payload. This can be anything. For example, in UNIX, a command shell can be spawned. For example, with /bin/sh in Windows NT/2000 and a specific Dynamic Link Library (DLL), external stacks may be preferable and may be used for further probing. For example, WININET.DLL can be used to send requests to and get information from the network and to download code or retrieve commands to execute.

The attacker may launch a denial-of-service attack or he or she may use the system as a launching point (ARP spoofing). The common attack is to spawn a remote shell. The exploited system can be converted into a covert channel or it can simulate Netcat to make a raw, interactive connection. The payload can be a worm that replicates itself and searches for fresh targets. The attacker can also eventually install a rootkit and remain in stealth mode after gaining super-user access.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



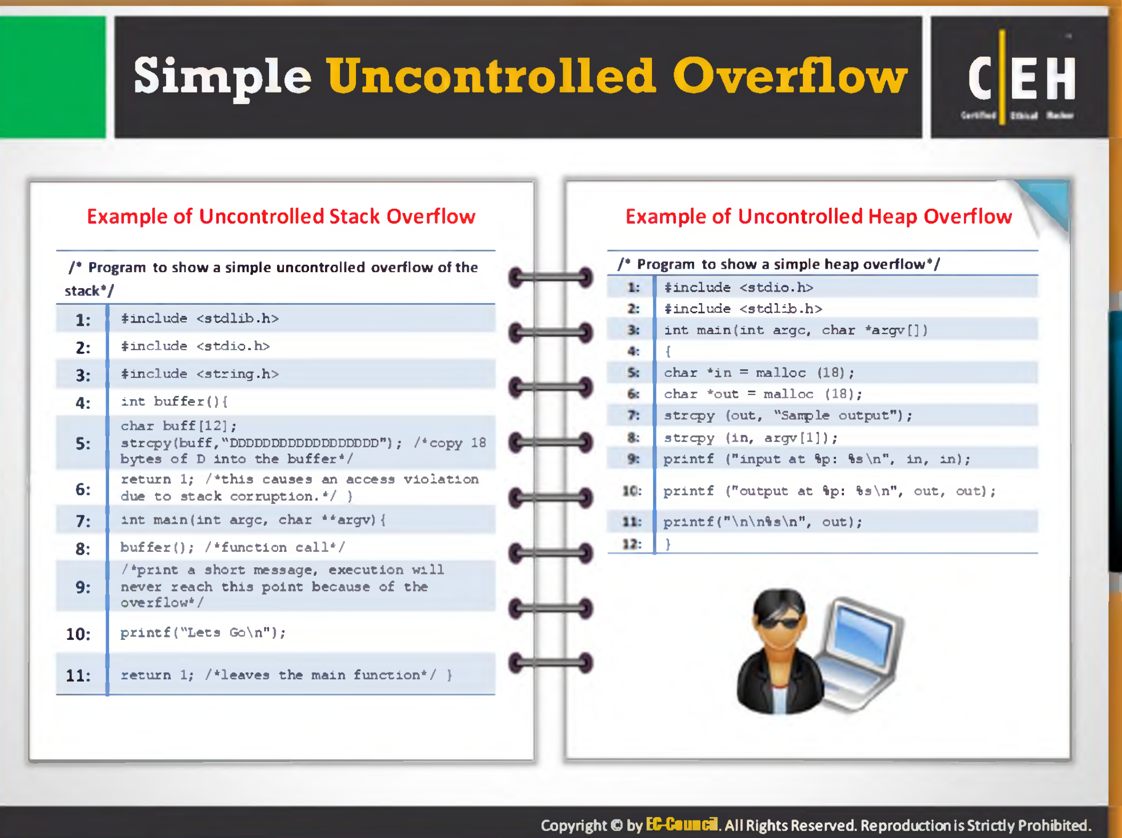
Module Flow

So far, we have discussed buffer overflow concepts and the methodology. Now it's time to see buffer overflow examples.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

This section covers various buffer overflow examples.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Simple Uncontrolled Overflow

Example of Uncontrolled Stack Overflow

```
/* stack3.c

This is a program to show a simple uncontrolled overflow of the stack. It
will overflow EIP with 0x41414141, which is AAAA in ASCII.

*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int bof()

{

      char buffer[8]; /* an 8 byte character buffer */

      /*copy 20 bytes of A into the buffer*/

      strcpy(buffer,"AAAAAAAAAAAAAAAAAAAA");

      /*return, this will cause an access violation due to stack corruption.
      We also take EIP*/

      return 1;
```


Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

```
}
int main(int argc, char **argv)
{
      bof(); /*call our function*/
      /*print a short message, execution will never reach this point because
      of the overflow*/
      printf("Not gonna do it!\n");
      return 1; /*leaves the main function*/
}
```

The main function in this program calls the bof() function. In the first line of bof() **function code** a buffer of char type with 8-bit size is initiated. Later a string of 20 As is copied into the buffer. This causes the **buffer overflow** because the size of buffer is just 8 bits,whereas the string copied into the buffer is 20 bits. This leads to an **uncontrolled overflow**.

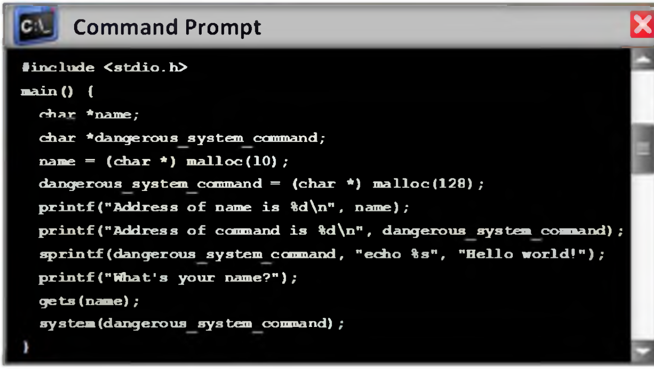
Example of Uncontrolled Heap Overflow

The following code is an example of uncontrolled head overflow.

```
/*heap1.c – the simplest of heap overflows*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
char *input = malloc (20);
char *output = malloc (20);
strcpy (output, "normal output");
strcpy (input, argv[1]);
printf ("input at %p: %s\n", input, input);
printf ("output at %p: %s\n", output, output);
printf("\n\n%s\n", output);
}
```


Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Simple Buffer Overflow in C



- The first thing the program does is declare two string variables and assign memory to them
- The "name" variable is given 10 bytes of memory (which will allow it to hold a 10-character string)
- The "`dangerous_system_command`" variable is given 128 bytes
- You have to understand that, in C, the memory chunks given to these variables will be located directly next to each other in the virtual memory space given to the program

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Simple Buffer Overflow in C

To understand how buffer overruns works, you need to look at the small C program that follows:

```
#include <stdio.h>
main() {
  char *name;
  char *dangerous_system_command;
  name = (char *) malloc(10);
  dangerous_system_command = (char *) malloc(128);
  printf("Address of name is %d\n", name);
  printf("Address of command is %d\n", dangerous_system_command);
  sprintf(dangerous_system_command, "echo %s", "Hello world!");
  printf("What's your name?");
  gets(name);
  system(dangerous_system_command);
}
```


Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

This program is designed to be run by a user on a console, but it illustrates the trouble that a poorly written **network daemon** can cause.

The first thing the program does is to declare two string variables, and assign memory to them. The "name" variable is given 10 bytes of memory (which will allow it to hold a 10-character string). The "**dangerous_system_command**" variable is given 128 bytes. The thing you have to understand is that in C, the memory chunks given to these variables will be located directly next to each other in the **virtual memory space** given to the program. If you run the program with a short name, you can see how things are supposed to work:

```
[jturner@secure jturner]$ ./overrun
Address of name is 134518696
Address of command is 134518712
What's your name?James
Hello world!
[jturner@secure jturner]$
```

As you can see, the address given to the "dangerous_system_command" variable is 16 bytes from the start of the "name" variable. The extra 6 bytes are overhead used by the "malloc" system call to allow the memory to be returned to **general usage** when it is freed.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Simple Buffer Overflow in C: Code Analysis

- The "gets" command, which reads a string from the standard input to the specified memory location, does not have a "length" specification
- This means it will read as many characters as it takes to get to the end of the line, even if it overruns the end of the memory allocated
- Knowing this, an attacker can overrun the "name" memory into the "dangerous_system_command" memory, and run whatever command he or she wishes

To compile the overrun.c program, run this command in Linux:

```
gcc overrun.c -o overrun
[XX]$ ./overrun
Address of name is 134518696
Address of command is 134518712
What's your name?xmen
Hello world!
[XX]$
```

The address given to the "dangerous_system_command" variable is 16 bytes from the start of the "name" variable

The extra 6 bytes are overhead used by the "malloc" system call to allow the memory to be returned to general usage when it is freed

Buffer Overrun Output

```
[XX]$ ./overrun
Address of name is 134518696
Address of command is 134518712
What's your
name?0123456789123456cat/etc/passwd

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Simple Buffer Overflow in C: Code Analysis

After allocating the memory and printing the **memory locations** of the two variables, the program generates a **command** that will later be sent to the "**system**" call, which causes it to be executed as if it had been typed at a keyboard. In this case, all it does is print "Hello world!" Then, it prompts the user for his or her name and reads it using the "gets" system call. In a real **network daemon**, this might be printing a prompt and **awaiting** a command from the client program such as a website address or email address.

The important thing to know is that "**gets**," which reads a string from standard input to the specified memory location, DOES NOT have a "length" specification. This means it will read as many characters as it takes to get to the end of the line, even if it **overruns** the end of the memory allocated. Knowing this, a hacker can overrun the "name" memory into the "**dangerous_system_command**" memory, and run whatever command they wish. For example:

[jturner@secure jturner]$ ./overrun

Address of name is 134518696

Address of command is 134518712

What's your name?0123456789123456cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

bin:x:1:1:bin:/bin:

daemon:x:2:2:daemon:/sbin:

adm:x:3:4:adm:/var/adm:

lp:x:4:7:lp:/var/spool/lpd:

sync:x:5:0:sync:/sbin:/bin/sync

shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown

halt:x:7:0:halt:/sbin:/sbin/halt

mail:x:8:12:mail:/var/spool/mail

By **padding** out the response to the name query to 16 character and then adding a system command, the system command overwrites "echo Hello World!" with "cat /etc/passwd". As you can see, this causes that command to be run instead of the appropriate one.

So what can be done to prevent this? First, use the fgets system call, which specifies a maximum length, will eliminate the possibility altogether. By changing the "gets" call to:

fgets(name, 10, stdin);

The problem is solved:

[jturner@secure jturner]$ ./overrun

Address of name is 134518768

Address of command is 134518784

What's your name?01234567890123456cat /etc/passwd

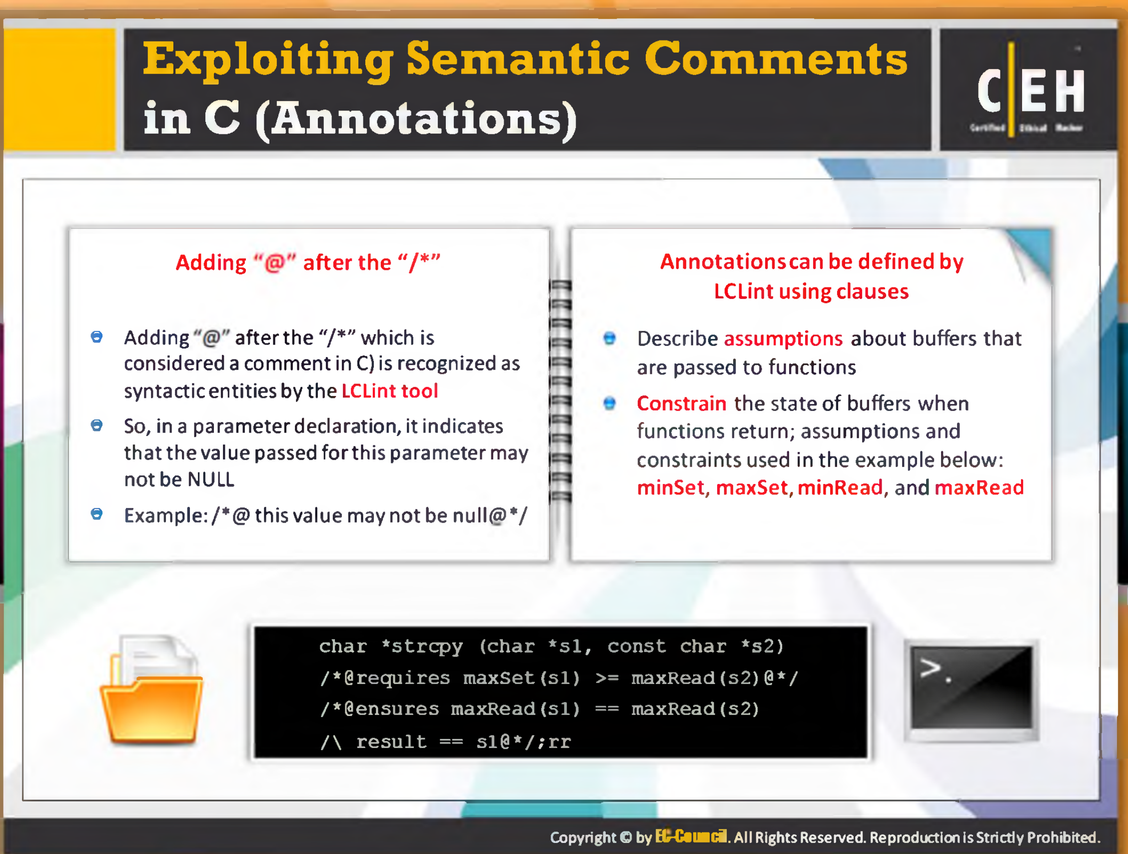
Hello world!

[jturner@secure jturner]$

But, since many sites run software that they do not have source code to (commercial databases, for example), you cannot protect yourself from all **buffer overruns.** The other important step you need to take is to turn off any network services you do not use, and only run the ones you do use at a **permission** level that meets the needs of the program. For example, do not run a **database** as root; give it its own user and group. That way, if it is exploited, it cannot be used to take over the system.

Buffer overruns are one of those things that every first-year **programming** student should be taught to avoid. That attackers still use it with such **frequency** is an indication of how far we have to go in the quest for truly **reliable** and **secure software**.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Exploiting Semantic Comments in C (Annotations)

Though many run-time approaches have been proposed to mitigate the risk associated with buffer overflows, they are not widely used. Hence, static analysis of a program's source code came into practice to detect buffer overflows. This can be accomplished with the help of the LCLint tool. It performs static detection of buffer overflows by exploiting semantic comments added to the source code. Thus, it enables local checking of interprocedural properties.

Adding "@" after the "/*"

- Adding "@" after the "/*" which is considered a comment in C) is recognized as syntactic entities by the LCLint tool
- So, in a parameter declaration, it indicates that the value passed for this parameter may not be NULL
- Example: /*@ this value need not be null@*/

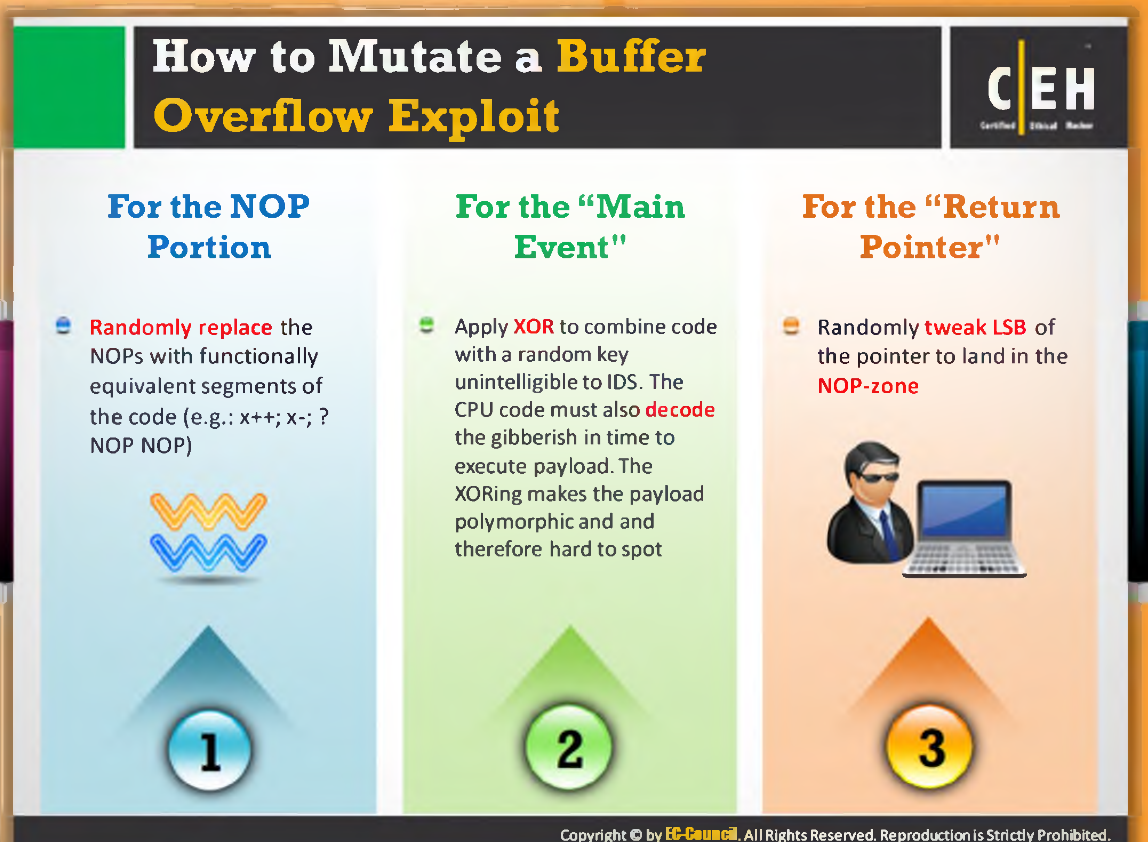
Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Annotations can be defined by LCLint using clauses

- Describe **assumptions** about buffers that are passed to functions
- Constrain the state of buffers when functions return **assumptions** and **constraints** used in the example below: minSet, maxSet, minRead, and maxRead

```
char  , const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
/\ result == s1@*/;rr
```


Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

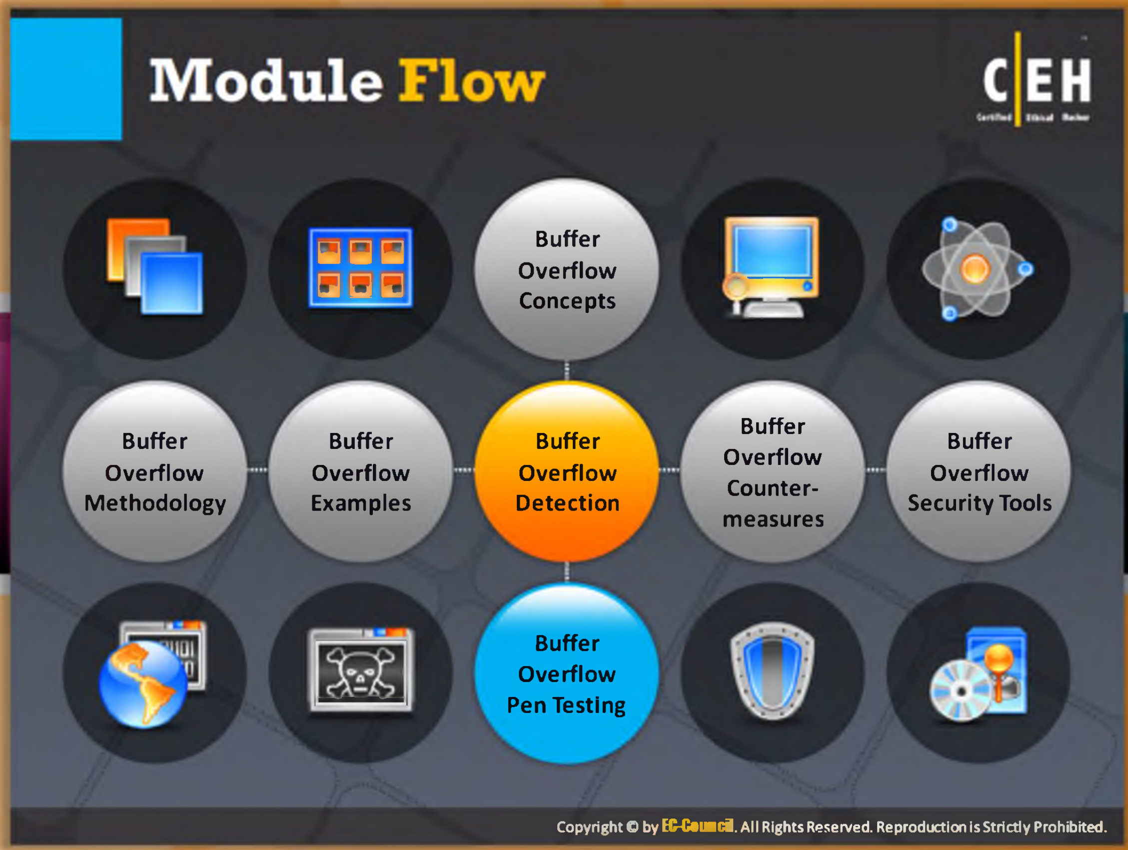


How to Mutate a Buffer Overflow Exploit

Most IDSes look for signs of NOP sleds. Detecting an array of NOPs can be indicative of a buffer overflow exploit across the network. ADMutate takes the concept a bit further. It accepts a buffer overflow exploit as input and randomly creates a functionally equivalent version (polymorphism, part deux).

ADMutate substitutes the conventional NOPs with operationally inert commands. ADMutate encodes the shellcode with a simple mechanism (xor) so that the shellcode will be unique to any NIDS sensor. This allows it to bypass shellcode signature analysis. XORing encodes the shellcode with a randomly generated key. It modulates the return address and the least significant byte is altered to jump into different parts of the stack. It also allows the attacker to apply different weights to generate ASCII equivalents of machine language code and to tweak the statistical distribution of resulting characters. This formulates the traffic as the "standard" for a given protocol, from a statistical perspective, for example, more heavily weighted characters "<" and ">" in HTTP protocol. To further reduce the pattern of the decoder, out-of-order decoders are supported. This allows the user to specify where in the decoder certain operational instructions are placed. ADMutate was developed to offend IDS signature checking by manipulation of buffer overflow exploits. It uses techniques borrowed from virus creators and works on Intel, Sparc, and HPPA processors. The likely targets are Linux, Solaris, IRIX, HPUX, OpenBSD, UnixWare, OpenServer, TRU64, NetBSD, and FreeBSD.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



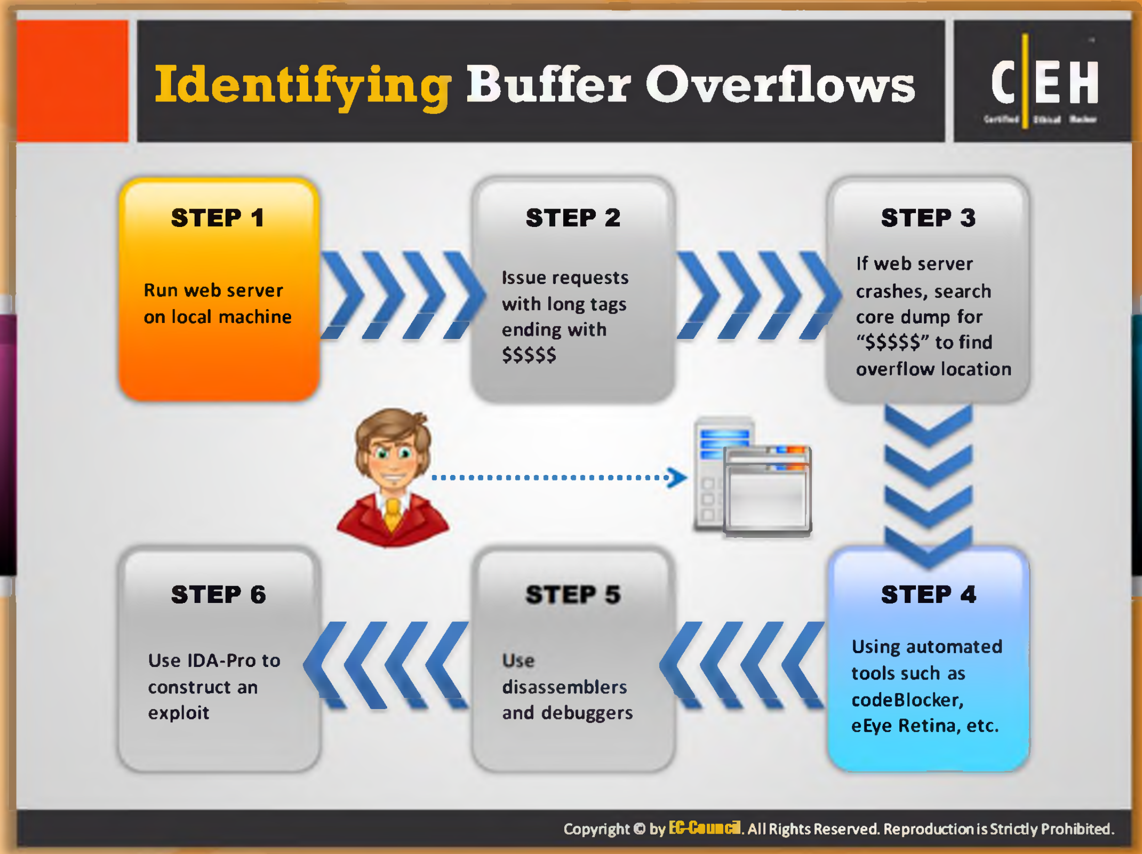
Module Flow

So far, we have discussed what buffer overflow is and how to exploit it. Now it's time to discuss how to detect buffer overflows.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

This section focuses on various buffer overflow detection methods such as testing for heap and stack overflows, formatting string conditions, and buffer overflow detection tools.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

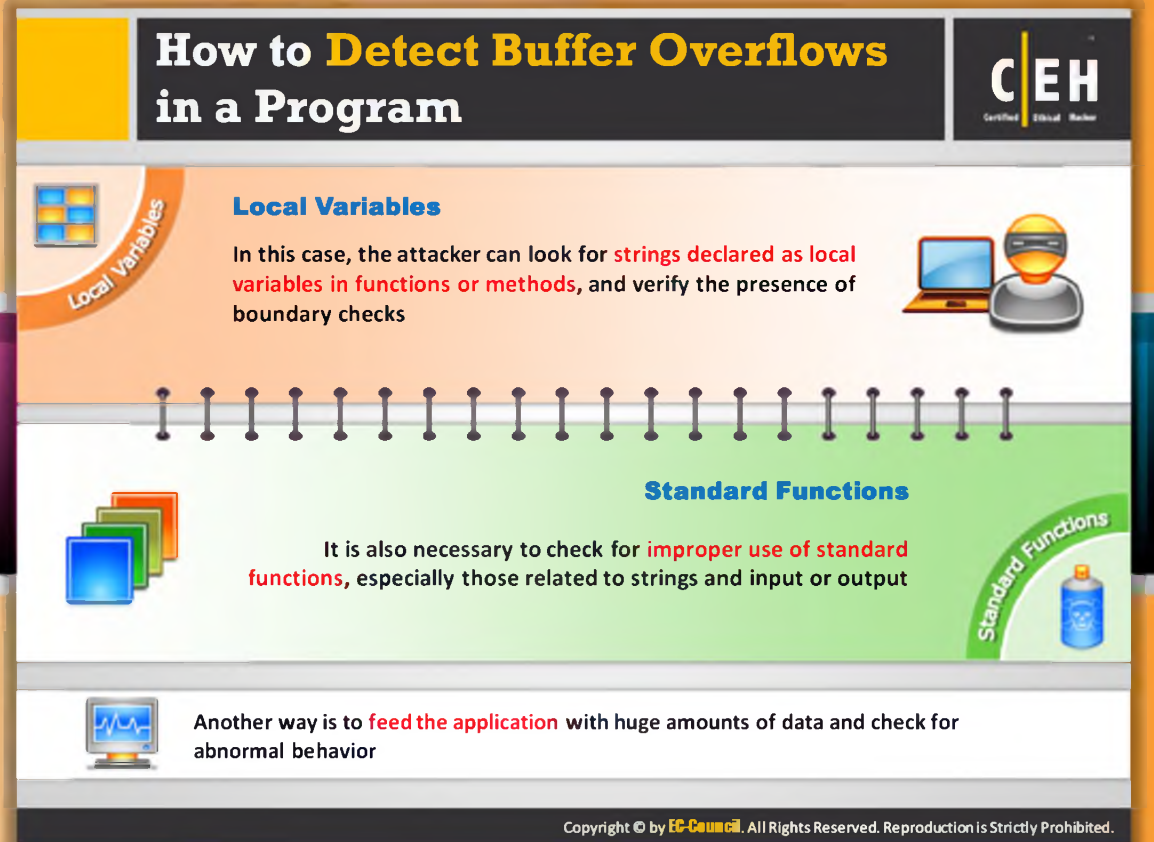


Identifying Buffer Overflows

In order to identify the buffer overflow vulnerability, follow the steps mentioned as follows:

- Step 1: Run web server on local machine
- Step 2: Issue requests with long tags-all long tags end with "$$$$$"
- Step 3: If the web server crashes, search core dump for "$$$$$" to find overflow location
- Step 4: Using automated tools such as codeBlocker, eEye Retina, etc.
- Step 5: Use disassemblers and debuggers
- Step 6: Use IDA-Pro to construct an exploit

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



How to Detect Buffer Overflows in a Program

To **identify buffer overflows**, you need to **examine** programs **systematically** in order to discover vulnerabilities. There are basically two main ways to detect buffer overflow **vulnerabilities**:

- The first method is to look at the source code:

  The attacker looks for strings declared as local variables in functions or methods and verifies the presence of a **boundary check** or use of "SAFE" C functions. In addition, it is also necessary to check for the improper use of standard functions, especially those related to strings and input/output.

- The second method is to feed the huge volumes of data to the application and check for abnormal behavior.

To start with, you can attempt to reverse the code using a **disassembler** or **debugger** and examine the code for vulnerabilities.

Disassembly starts from the entry point of the program and then proceeds with all routes of execution to search for the functions that are **external** to the main flow of the program. The user may keep his or her focus on functions lying outside main () and check those **subroutines** that take strings as their input or generate them as output.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

As already mentioned, programs written in C are particularly susceptible, because the language does not have any **built-in bounds checking**, and overflows are **undetected** as they write past the end of a character array. The standard C library offers many functions for the purpose of copying or appending strings that do not perform any **boundary check**. These include strcat (), strcpy (), sprintf (), and vsprintf (). These functions operate on **null-terminated** strings and do not check for an overflow resulting from a received string.

The gets () function reads a line from stdin into a buffer until a terminating new line or EOF occurs. It does not check for any buffer overflows. The **scanf()** function also gives rise to potential overflows, if the program attempts to match a sequence of non-white-space characters (%s) or a **non-empty sequence** of characters from a specified set (%[ ]).

The array pointed to by the char pointer is inadequate to accept the entire sequence of characters, and the optional maximum field width is not specified. If the target of any of these functions is a buffer of static size, and its arguments are derived from user input, there is a good chance of encountering a buffer overflow.

Most attackers point out that **ingenuity** is critical for **exploiting** buffer overflow vulnerabilities. This is true especially when one has to guess a few parameters. For instance, if the attacker is looking at software that assists in communication such as FTP, he or she may be looking at commands that are typically used and how they are implemented.

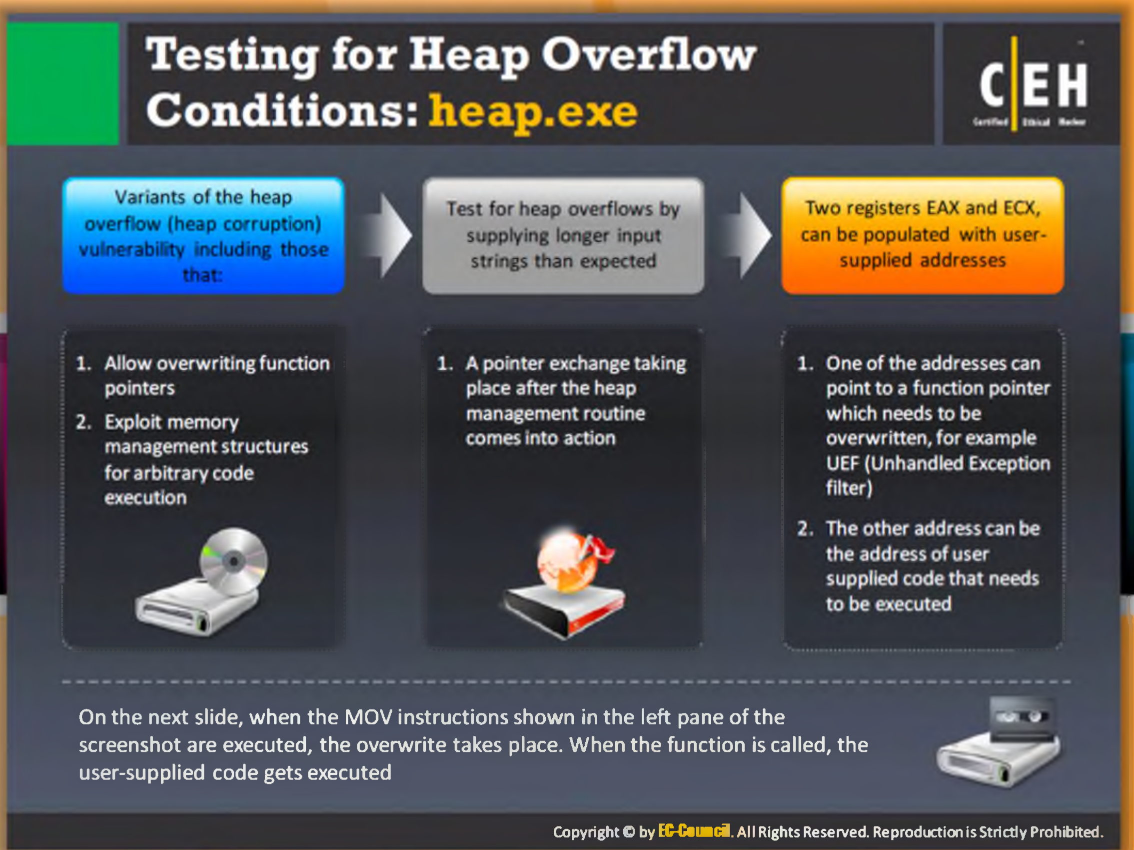
For example, the attacker can search for text and pick out a suspect variable from a table. He or she can then go on and check the code for any boundary checks and functions such as strcpy() that take input directly from the buffer. The **emphasis** can be on local variables and parameters. The attacker can then test the code by providing **malformed** input and observe the resulting behavior of the code.

Another method is to adopt a brute force approach by using an automated tool to **bombard** the program with excessive amounts of data and cause the program to crash in a meaningful way. The attacker can then examine the register dump to check whether the data bombarding the program made its way into the instruction pointer.

What happens after the buffer overflow vulnerability is discovered? After discovering a vulnerability, the attacker can observe carefully how the call obtains its user input and how it is routed through the function call. He or she can then write an exploit, which makes the software do things it would not do normally. This can range from simply crashing the machine to injecting code so that the attacker can gain remote access to the machine. He or she might then use the **compromised system** as a launch base for further attacks.

However, the greatest threat comes from a **malicious program** such as a worm that is written to take advantage of the **buffer overflow**.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Heap Overflow Conditio ns: heap.exe

A heap is memory that is **allocated dynamically**; these are dynamically removed (example delete, free) and created (example new, malloc). In some cases, heaps are reallocated by the programmer. Each memory chunk in a heap is associated with **boundary tags** containing information about **memory management**.

Heap-based buffer overflow causes overwriting the **control information**. This leads to an access violation when the heap management routine frees the buffer. This overflow vulnerability allows an attacker to overwrite a desired memory location with a user-controlled value, when executed in a controlled fashion. Thus, it allows an attacker to **overwrite function** pointers and other addresses stored in TEB, GOP, or .dtors structures with the shellcode's address.

There are many ways in which the heap overflow vulnerability can be **exploited** to **execute shellcode** by overwriting function pointers. In order to exploit these vulnerabilities, certain conditions need to exist in the code. Hence, identifying or locating these **vulnerabilities** requires closer examination when compared to **stack overflow vulnerabilities**.

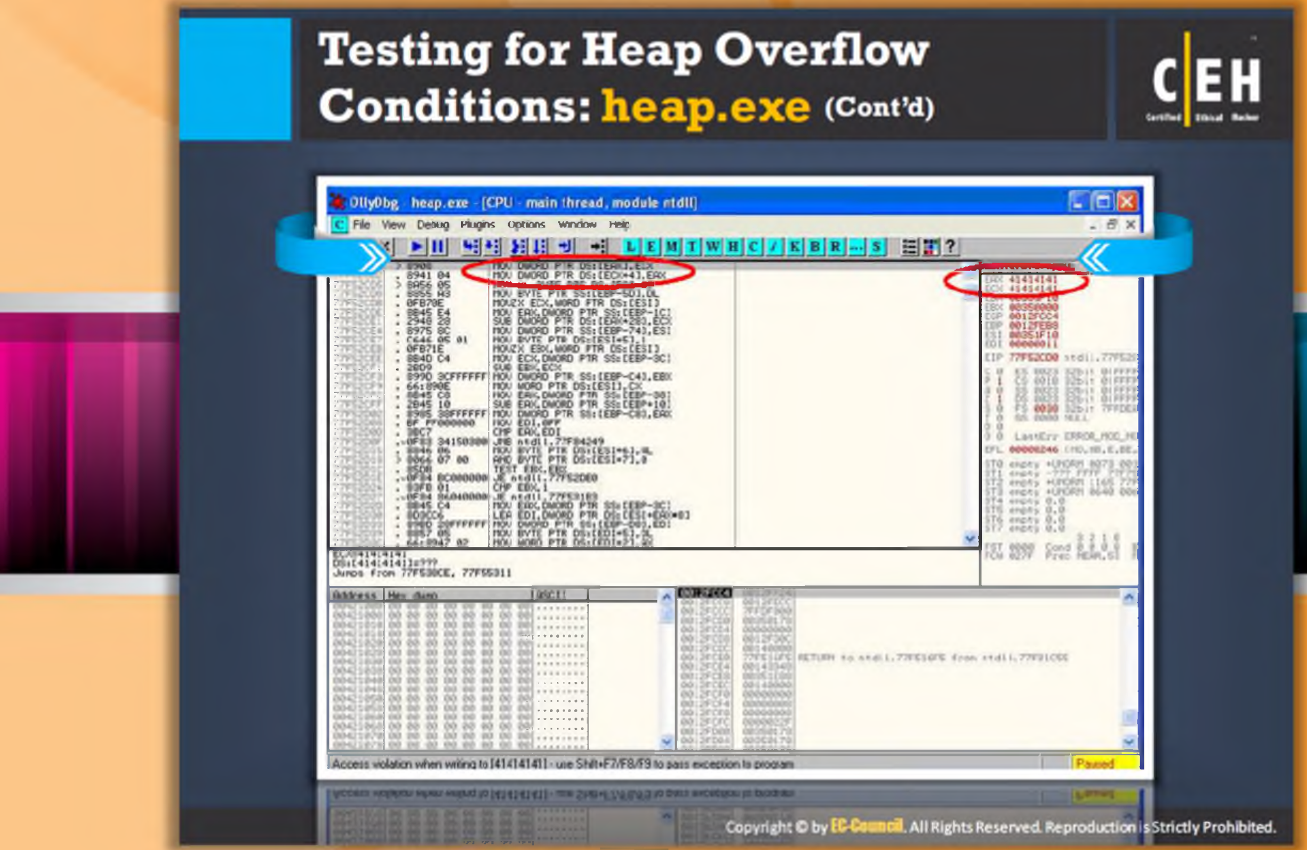
You can test for **heap overflows** by supplying input strings longer than expected. **Heap overflow** in a Windows program may appear in various forms. The most common one is pointer exchange taking place after the **heap management routine** frees the buffer.

Two registers EAX and ECX, can be populated with user-supplied addresses:

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

- One of the addresses can point to a function pointer that needs to be overwritten, for example, UEF (Unhandled Exception filter)
- The other address can be the address of user-supplied code that needs to be executed

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Heap Overflow Conditions: heap.exe (Cont'd)

Let us consider the following example of heap overflow vulnerability:

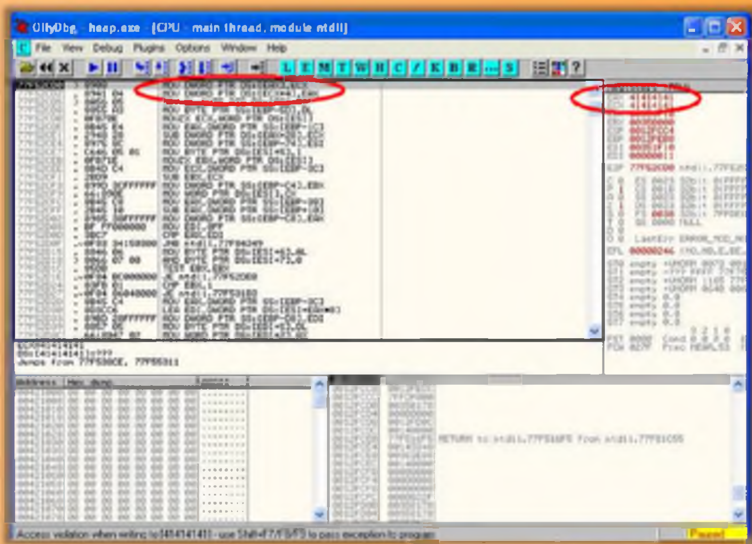


FIGURE 18.9: Testing for Heap Overflow Conditions in heap.exe

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

The two registers shown, **EAX** and **ECX**, can be populated with user-supplied addresses used to overflow the heap buffer. One address points to a function pointer such as **Unhandled Exception Filter (UEF)** that needs to be overwritten, and the other holds the address of the arbitrary code. The overwrite takes place when the **MOV instructions** are executed. When the function is called, the **arbitrary code** gets executed.

In addition to this method, the **heap-based buffer overflows** can be identified by **reverse engineering** the application binaries and using **fuzzing techniques**.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

Steps for Testing for Stack Overflow in OllyDbg Debugger

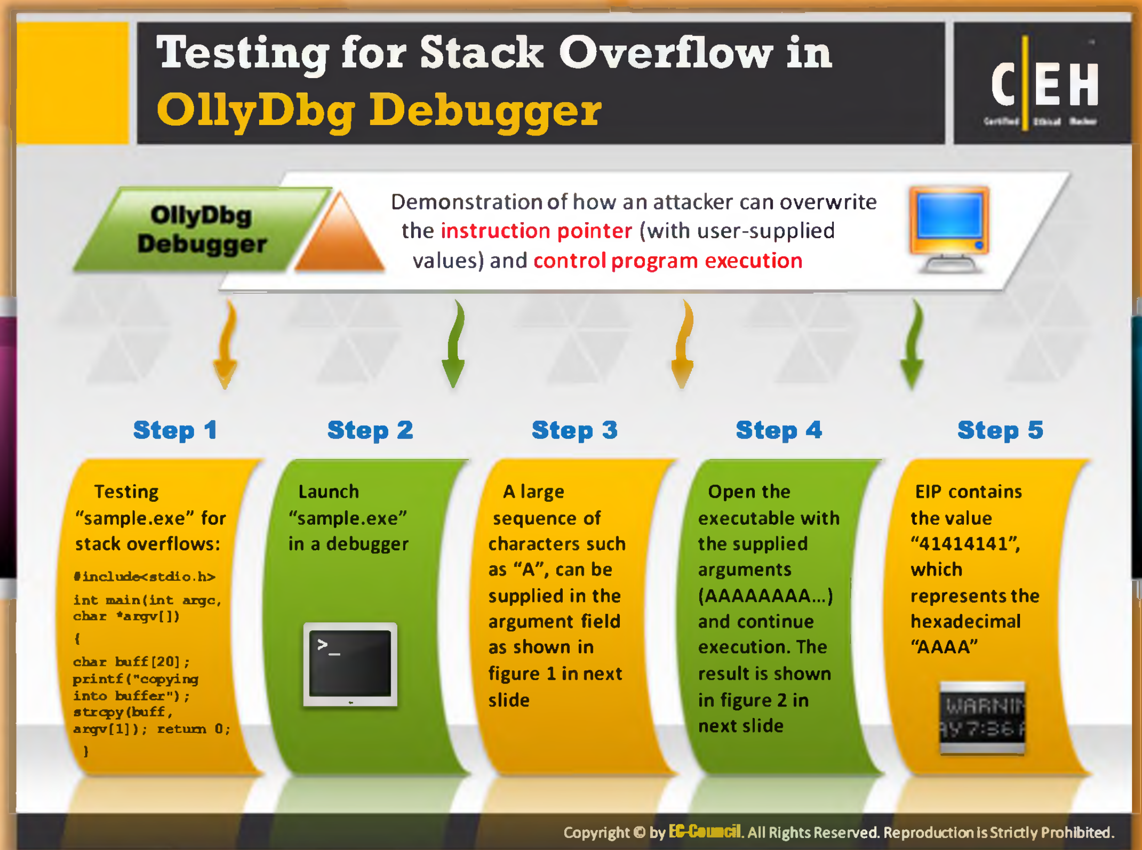
Stack buffer overflow occurs when the **variable data** size larger than the buffer size is placed in the program stack without bound checking. This can be a serious vulnerability and may even cause denial-of-service attacks. A **stack overflow vulnerability** allows an attacker to take control of the **instruction pointer**. This may lead to severe consequences. Therefore, you need to test your application or processes against stack overflow vulnerabilities.

Similar to **heap-based buffer overflow** testing, the stack overflow vulnerability can also be tested by supplying a large amount of input data than the normal or expected. However, this alone is not enough. In addition to sending a large amount of input data, you need to inspect the execution flow of the application and the responses to check whether an overflow has occurred or not. You can do this in four steps with the help of a **debugger**, a computer program used to test other programs. Here we are testing for stack overflow with the help of the **OllyDbg debugger**. The first step in testing for stack overflow is to attach a debugger to the target application or process. Once you successfully attach the program, you need to generate the **malformed** large input data for the target application to be tested. Now, supply the malformed input to the application and **inspect** the responses in a **debugger**. The ollydbg debugger allows you to observe the execution flow and the state of registers when the stack

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

overflow gets **triggered**. On the next slide, we will discuss these steps by considering an example.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Stack Overflow in OllyDbg Debugger

Here we are going to demonstrate how an attacker can **overwrite** the instruction pointer (with user-supplied values) and **control** program execution. Consider the following example of "sample.exe" to test for stack overflows:

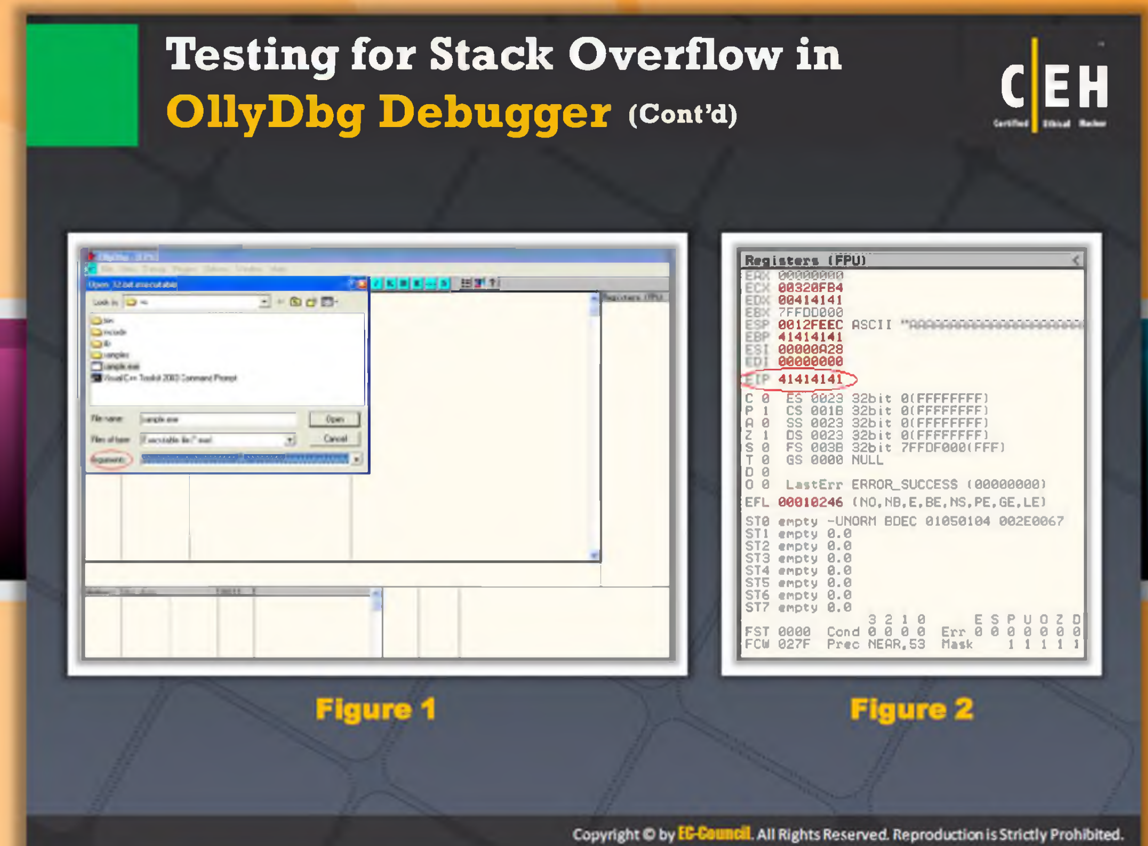
```
#include<stdio.h>
int main(int argc,char *argv[])
{
char buff[20]; printf("copying into buffer");
strcpy(buff, argv[1]);
return 0;
 }
```

Launch the sample.exe in OllyDbg debugger. The sample.exe accepts **command line arguments**. So you can supply a large **sequence of characters** such as 'A' in the **argument field** as shown in figure 1 on the next slide. Now open sample.exe with the supplied arguments (AAAAAAAAA....) and continue execution. The result is shown in figure 2 on the next slide. From the figure 2 in

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

next slide, it is clear that the EIP (Extended Instruction Pointer) contains the value "41414141." The hexadecimal representation of character 'A' is 41. Hence "41414141" represents the string "AAAA."

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Stack Overflow in OllyDbg Debugger (Cont'd)

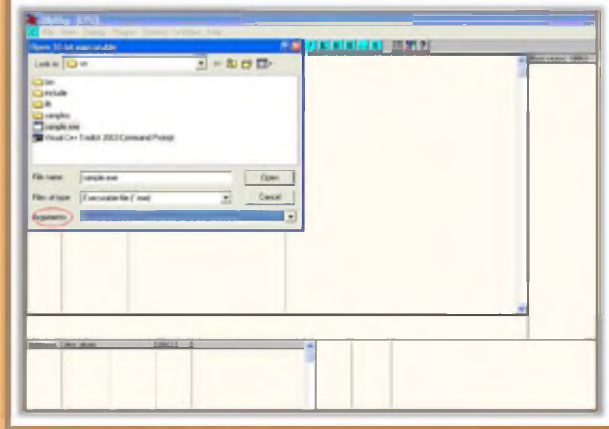
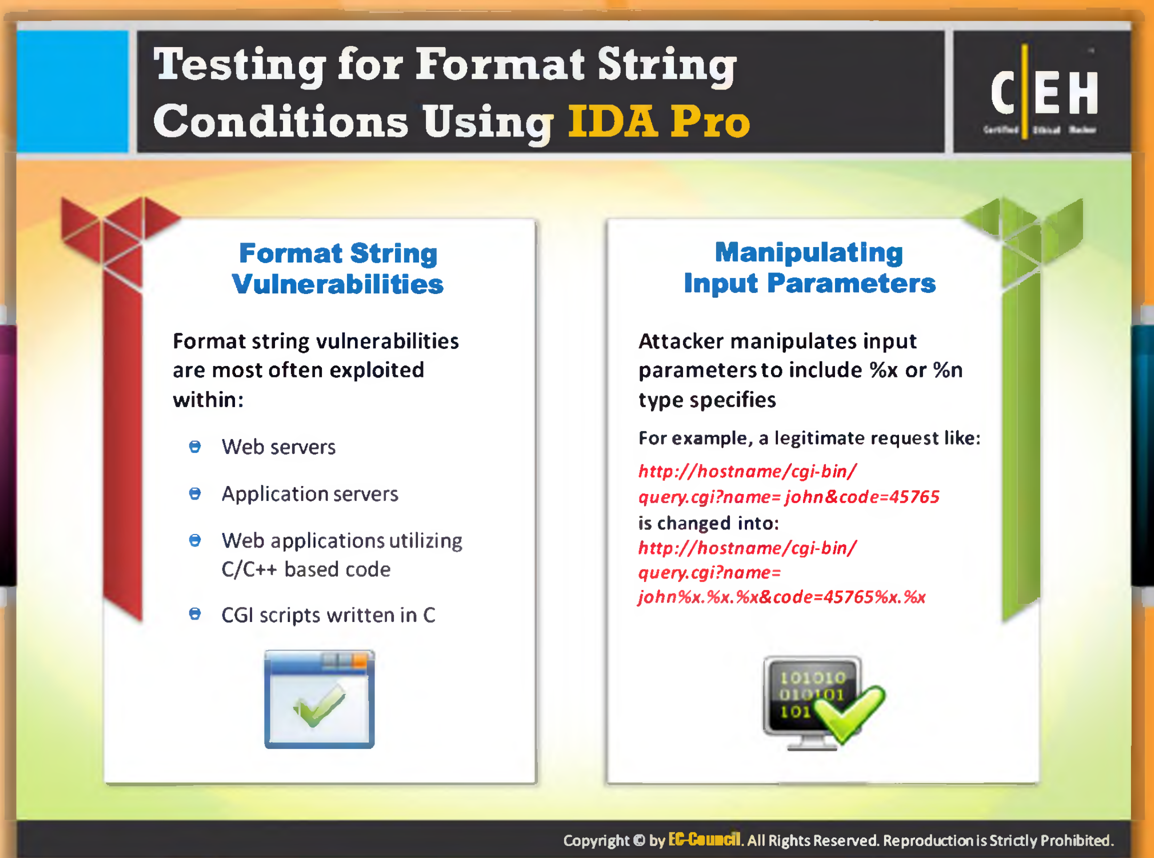


Figure 1 Figure 2

FIGURE 18.10: Testing for Stack Overflow in OllyDbg Debugger

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Format String Conditions Using IDA Pro

Applications can be tested for format **string vulnerabilities** by supplying format type specifiers in application input. Format string vulnerabilities usually appear in **web servers**, **application servers**, or web applications utilizing **C/C++** based code or **CGI scripts** written in C. Most of these format string vulnerabilities are resulting because of the **insecure** call to error **reporting** or **logging** function like syslog().

An attacker manipulates input parameters to include %x or %n type specifiers in a CGI script. Consider a legitimate example:

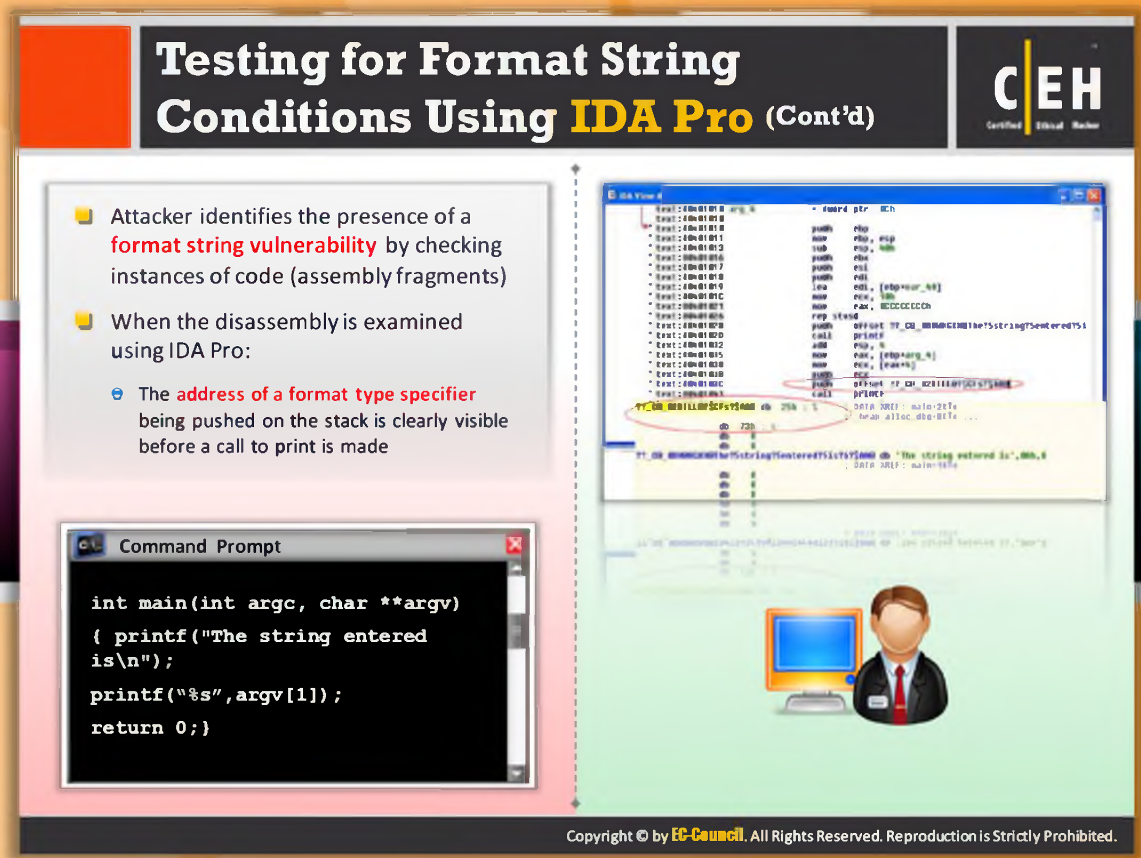
```
http://hostname/cgi-bin/query.cgi?name= john&code=45765
```

Attacker can manipulate this to

```
http://hostname/cgi-bin/query.cgi?name= john%x.%x.%x&code=45765%x.%x
```

If the routine processing the altered request contains any format string vulnerability, then it prints out the stack data to browser.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Testing for Format String Conditions Using IDA Pro (Cont'd)

An attacker identifies the presence of a format string vulnerability by checking instances of code (assembly fragments). Consider the following example code:

```
int main(int argc, char **argv)
{ printf("The string entered is\n");
printf("%s",argv[1]);
return 0;}
```

Examine the disassembly of the code using IDA Pro. You can clearly see the address of format type specifier being pushed on the stack before a call to printf is made.

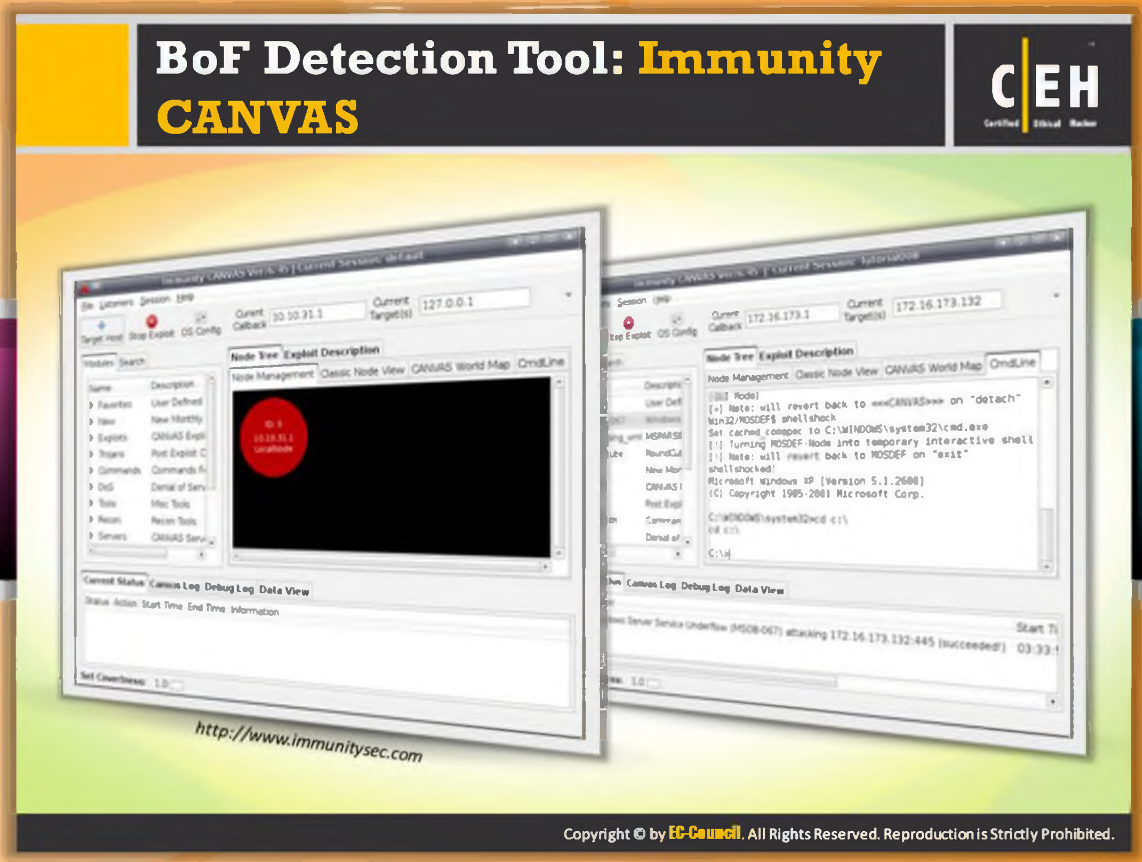
No offset will be pushed on the stack before calling printf, when the same code is compiled without "%s" as an argument.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

```
IDA View-A
text:00401010 arg_4          = dword ptr  0Ch
text:00401010
text:00401010                push    ebp
text:00401011                mov     ebp, esp
text:00401013                sub     esp, 40h
text:00401016                push    ebx
text:00401017                push    esi
text:00401018                push    edi
text:00401019                lea     edi, [ebp+var_40]
text:0040101C                mov     ecx, 10h
text:00401021                mov     eax, 0CCCCCCCCh
text:00401026                rep stosd
text:00401028                push    offset ??_C@_0BH@HGKH@The?5string?5entered?5i
text:0040102D                call    printf
text:00401032                add     esp, 4
text:00401035                mov     eax, [ebp+arg_4]
text:00401038                mov     ecx, [eax+4]
text:0040103B                push    ecx
text:0040103C                push    offset ??_C@_02DILL@?$CFs?$AA@
text:00401041                call    printf
??_C@_02DILL@?$CFs?$AA@ db  25h ; %      ; DATA XREF: main+2C↑o
                                         ; _heap_alloc_dbg+8C↑o ...
                db  73h ; s
                db    0
                db    0
??_C@_0BH@HGKH@The?5string?5entered?5is?6?$AA@ db 'The string entered is',0Ah,0
                                         ; DATA XREF: main+18↑o
                db    0
                db    0
                db    0
```

FIGURE 18.11: Testing for Format String Conditions Using IDA Pro

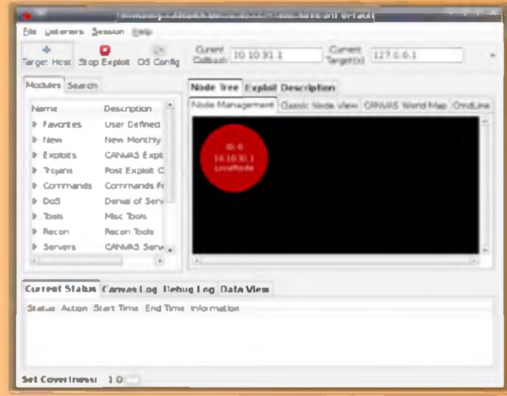
Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



BoF Detection Tool: Immunity CANVAS

Source: http://www.immunitysec.com

Immunity's CANVAS is an **exploit development** framework for **penetration testers** and **security professionals**. It allows you to discover how vulnerable you really are. It comes with packaged vulnerability exploitation modules for scripting and framework for developing **original exploits**. Thus, it provides a way for any organization to have a picture of their security posture, without guesswork or estimation.



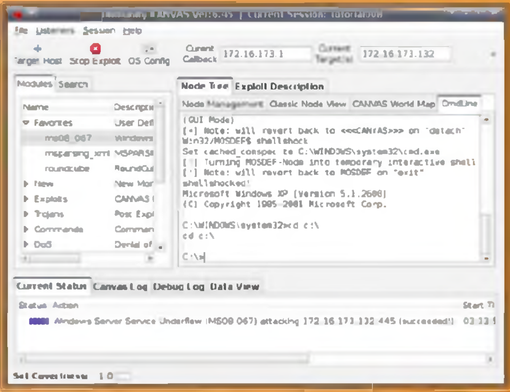
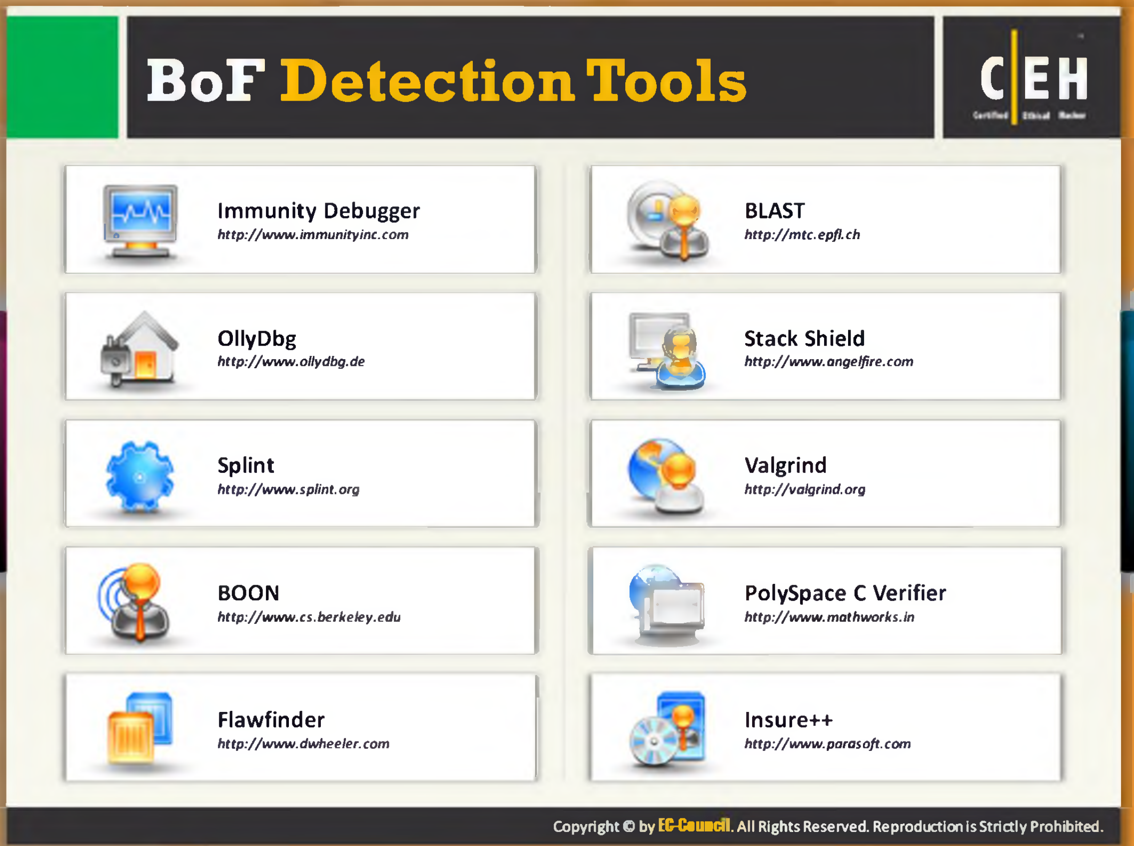


FIGURE 18.12: Immunity CANVAS Screenshot

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

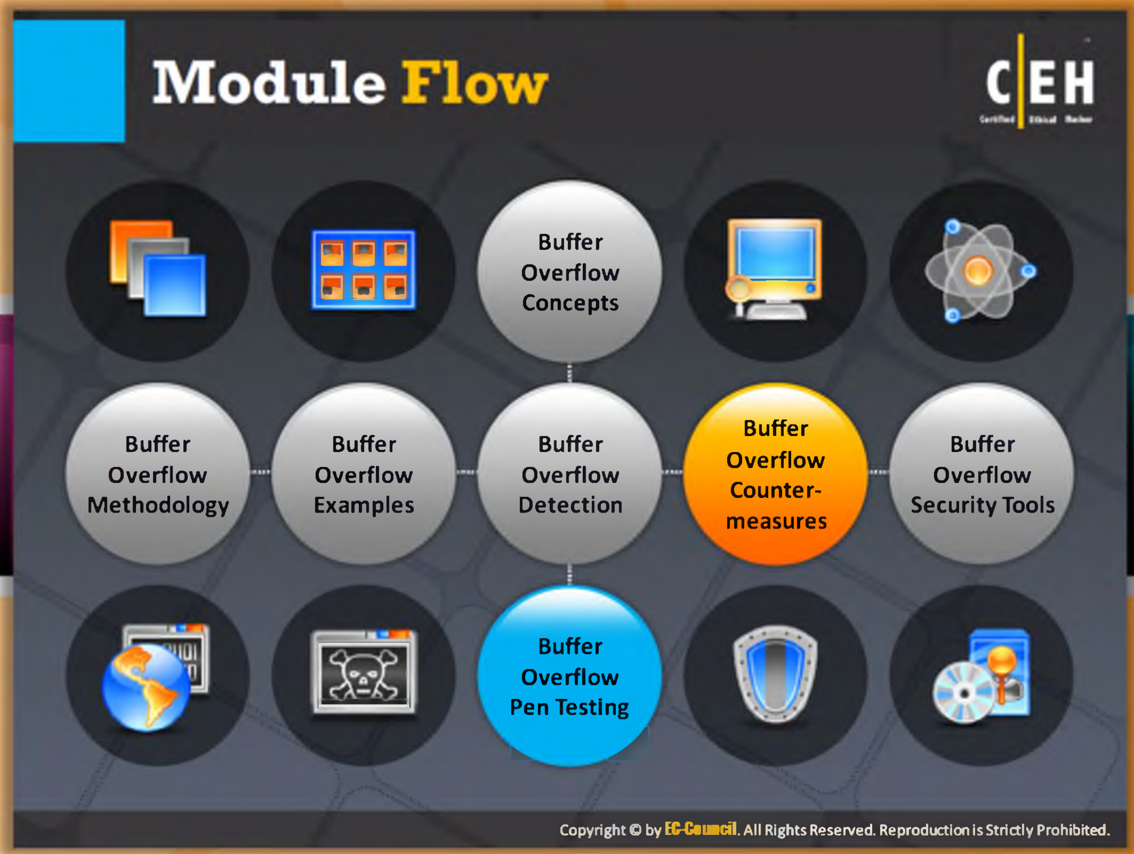


BoF Detection Tools

In addition to OllyDbg Debugger, IDA Pro, and Immunity CANVAS, many other tools have the capability to detect buffer overflows. A few buffer overflow detection tools are listed as follows:

- Immunity Debugger available at http://www.immunityinc.com
- OllyDbg available at http://www.ollydbg.de
- Splint available at http://www.splint.org
- BOON available at http://www.cs.berkeley.edu
- Flawfinder available at http://www.dwheeler.com
- BLAST available at http://mtc.epfl.ch
- Stack Shield available at http://www.angelfire.com
- Valgrind available at http://valgrind.org
- PolySpace C Verifier available at http://www.mathworks.in
- Insure++ available at http://www.parasoft.com

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Module Flow

So far, we have discussed the buffer overflow vulnerability, how to exploit it, and how to detect it. Once you detect buffer overflows, you should immediately apply or take countermeasures to protect your resources from being compromised. There are many reasons for buffer overflow exploits. The countermeasures to be applied may vary depending on the kind of buffer overflow vulnerability.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

This section suggests various **countermeasures** to different kinds of buffer overflow vulnerabilities. Thus, it can help you to **prevent buffer overflow** attacks.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

Defense against Buffer Overflows

The errors in programs are the main cause of **buffer flow problems**. These problems are responsible for **security vulnerabilities** using which the attacker tries to gain unauthorized access to a remote host. Attackers easily insert and execute attack code. To avoid such problems, some protection measures have to be taken. **Protection measures** to defend against buffer overflows include:

Manual auditing of code

Search for the use of **unsafe functions** in the C library like **strcpy()** and replace them with safe functions like strncpy(), which takes the size of the buffer into account. Manual auditing of the source code must be undertaken for each program.

Compiler techniques

Range checking of indices is defined as a defense that guarantees **100% efficiency** from buffer overflow attacks. Java automatically checks if an array index is within the proper bounds. Use compilers like Java, instead of C, to avoid buffer overflow attacks.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

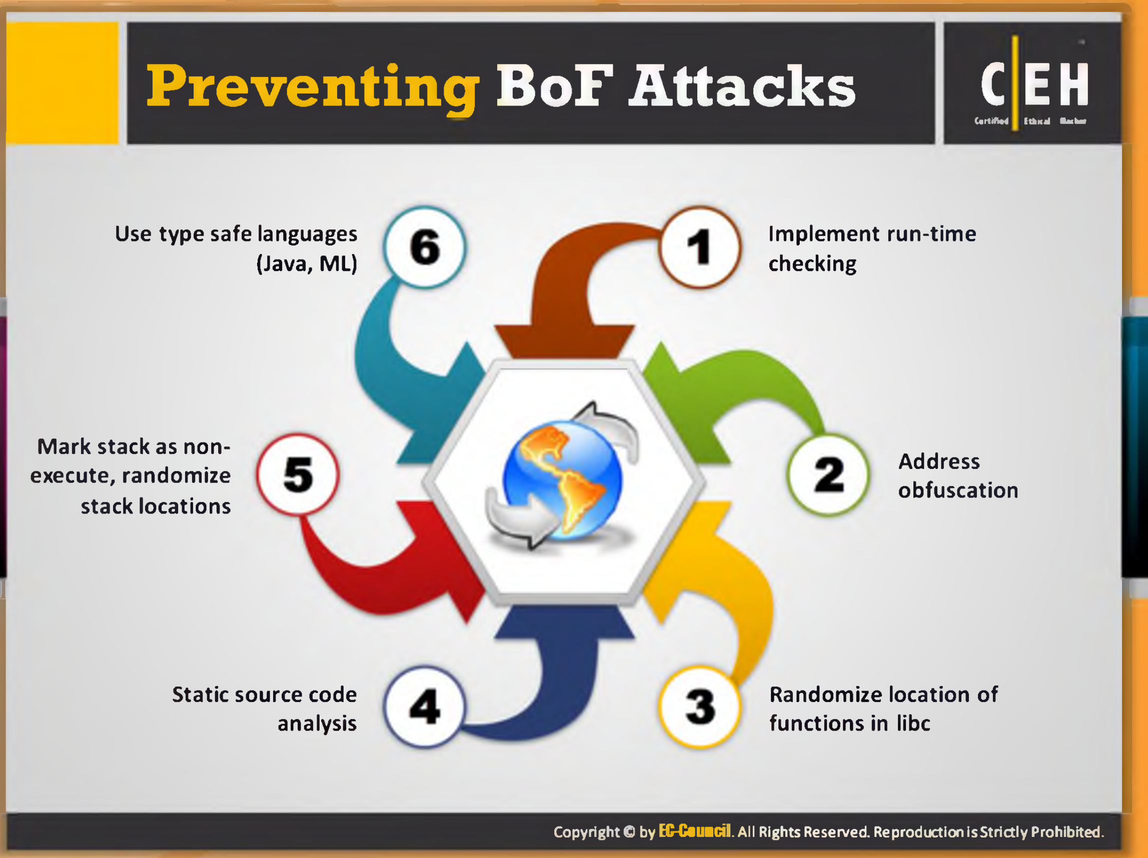
Safer C library support

A robust alternative is to provide safe versions of the **C library functions** where it attacks by overwriting the return address. It works with the binaries of the **target program's** source code and does not require access to the program's source code. It can be handled according to the occurrence of the threat without any **vendors operating** against it. It is available for Windows 2000 systems and is an **effective technique**.

Disabling stack execution

This is an easy solution that provides an option to install the **OS-disabling stack** execution. The idea is simple, inexpensive, and relatively effective against the current crop of attacks. A weakness in this method is that some programs depend on the execution of the stack.

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.

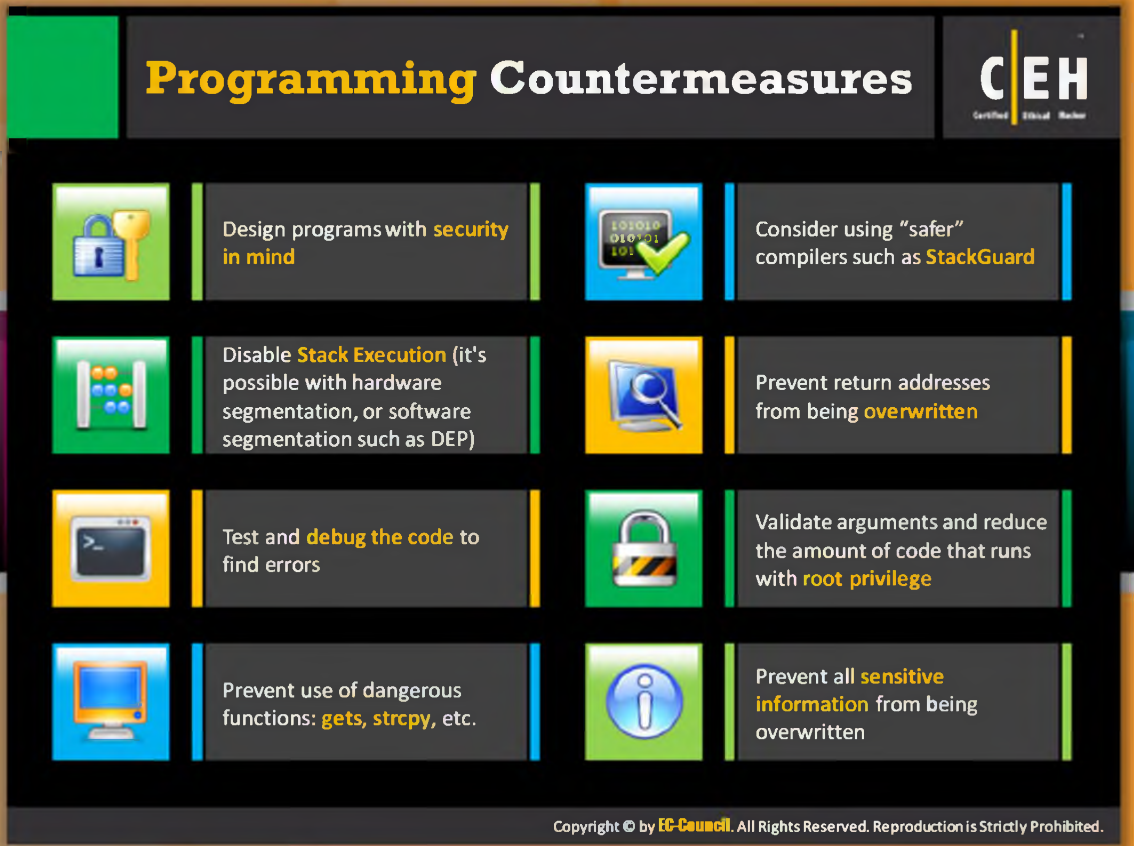


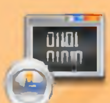
Preventing BoF Attacks

A buffer overflow attack occurs when large amounts of data are sent to the system, more than it is intended to hold. This attack usually occurs due to **insecure programming**. Often this may lead to a **system crash**. To avoid such problems, some **preventive measures** are adopted. They are:

- Implement run-time checkingAddress obfuscation
- **Randomize** location of functions in libcStatic source code analysis
- Mark stack as non-execute, random stack location
- Use type safe languages (Java, ML)

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.





Programming Countermeasures

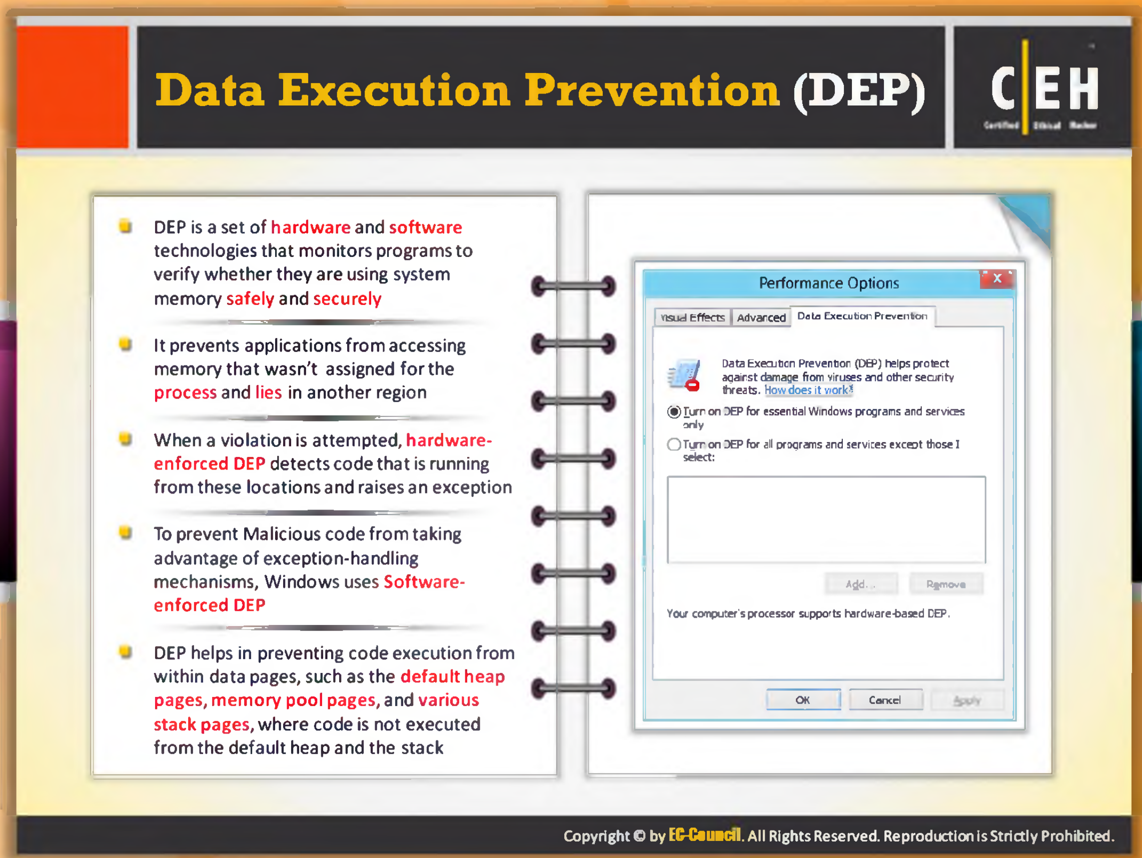
- Design programs with security in mind.
- Disable dtack execution (possible on Solaris).
- Test and debug the code to find errors.
- Prevent use of dangerous functions: gets, strcpy, etc.
- Consider using "safer" compilers such as StackGuard.
- Prevent return addresses from being overwritten.
- Validate arguments and reduce the amount of code that runs with root privilege.
- Prevent all sensitive information from being overwritten.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Programming Countermeasures (Cont'd)

- Make changes to the C language itself at the language level to **reduce the risk** of buffer overflows.
- Use static or **dynamic source code analyzers** at the source code level to check the code for buffer overflow problems.
- Change the compiler at the **compiler level** that does bounds checking or protects addresses from overwriting.
- Change the rules at the **operating system** level for which memory pages are allowed to hold executable data.
- Make use of safe libraries.
- Make use of tools that can **detect** buffer overflow vulnerabilities.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Data Execution Prevention (DEP)

Data execution prevention (DEP) is a set of **hardware** and **software technologies** that monitors programs to verify whether they are using system memory safely and securely. It prevents the applications that may access memory that wasn't assigned for the process and lies in another region. When an execution occurs, **hardware-enforced DEP** detects code that is running from these locations and raises an exception. To prevent malicious code from taking advantage of **exception-handling** mechanisms in Windows, use software-enforced DEP.

DEP helps in **preventing code execution** from data pages, such as the default heap pages, memory pool pages, and various stack pages, where code is not executed from the default heap and the stack.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

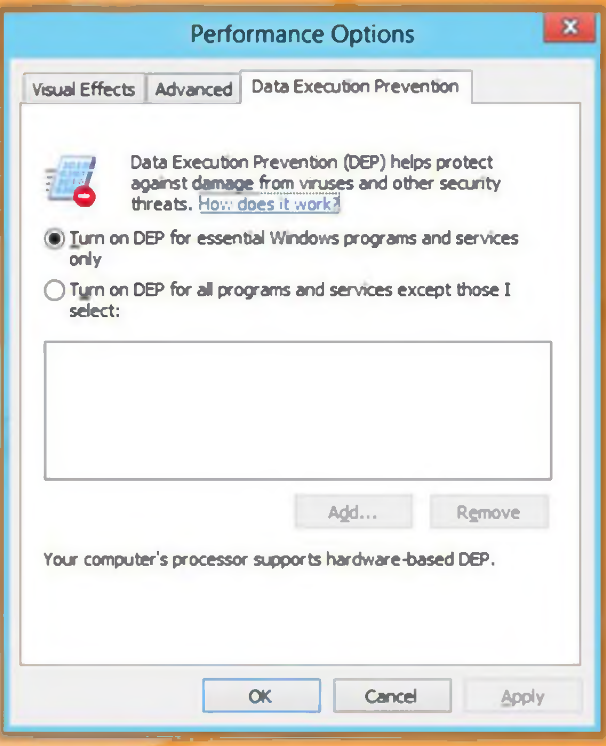
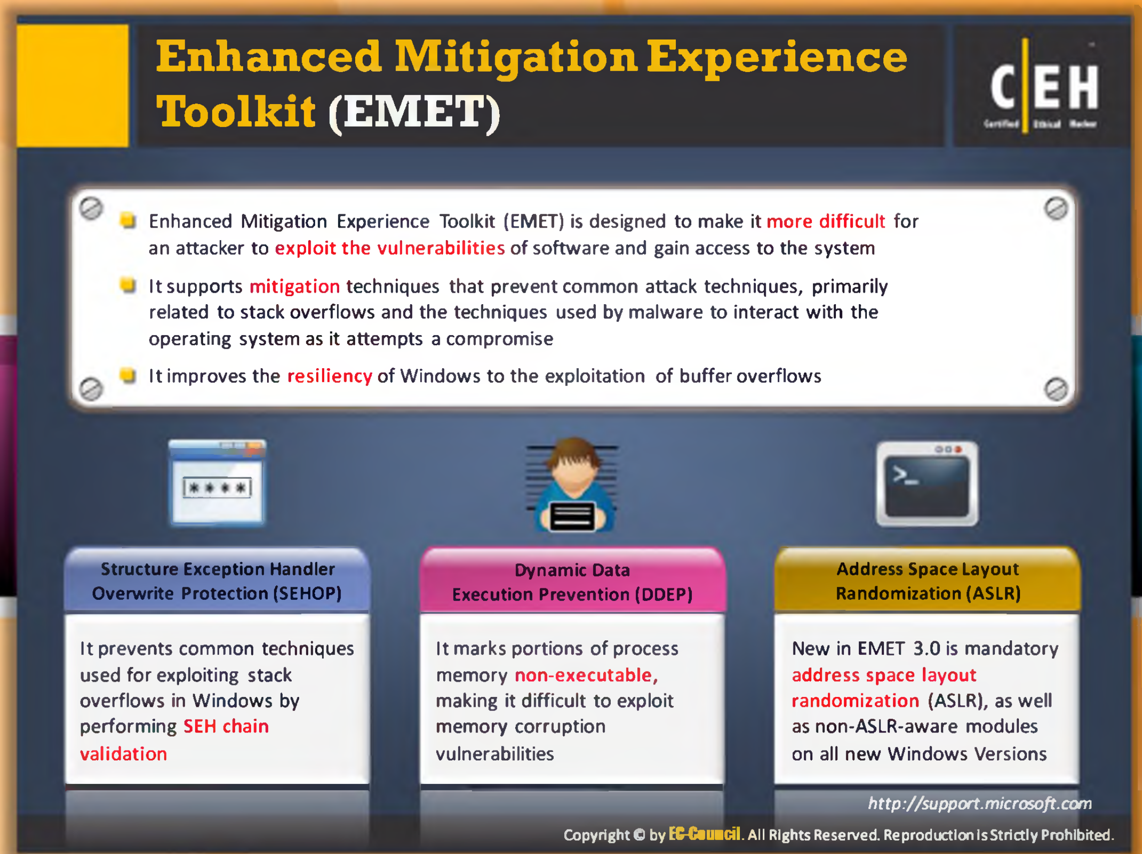


FIGURE 18.13: Data Execution Prevention (DEP)

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



Enhanced Mitigation Experience Toolkit (EMET)

Enhanced Mitigation Experience Toolkit (EMET) is designed to make it more difficult for an attacker to **exploit the vulnerabilities** of software and gain access to the system. It supports mitigation techniques that prevent **common attack techniques**, primarily related to **stack overflows** and the techniques used by malware to interact with the operating system as it attempts the compromise. It improves the **resiliency of Windows** to the exploitation of buffer overflows.

- **Structure Exception HandlerOverwrite Protection (SEHOP):**

  It prevents common techniques used for exploiting stack overflows in Windows by performing **SEH chain validation**.

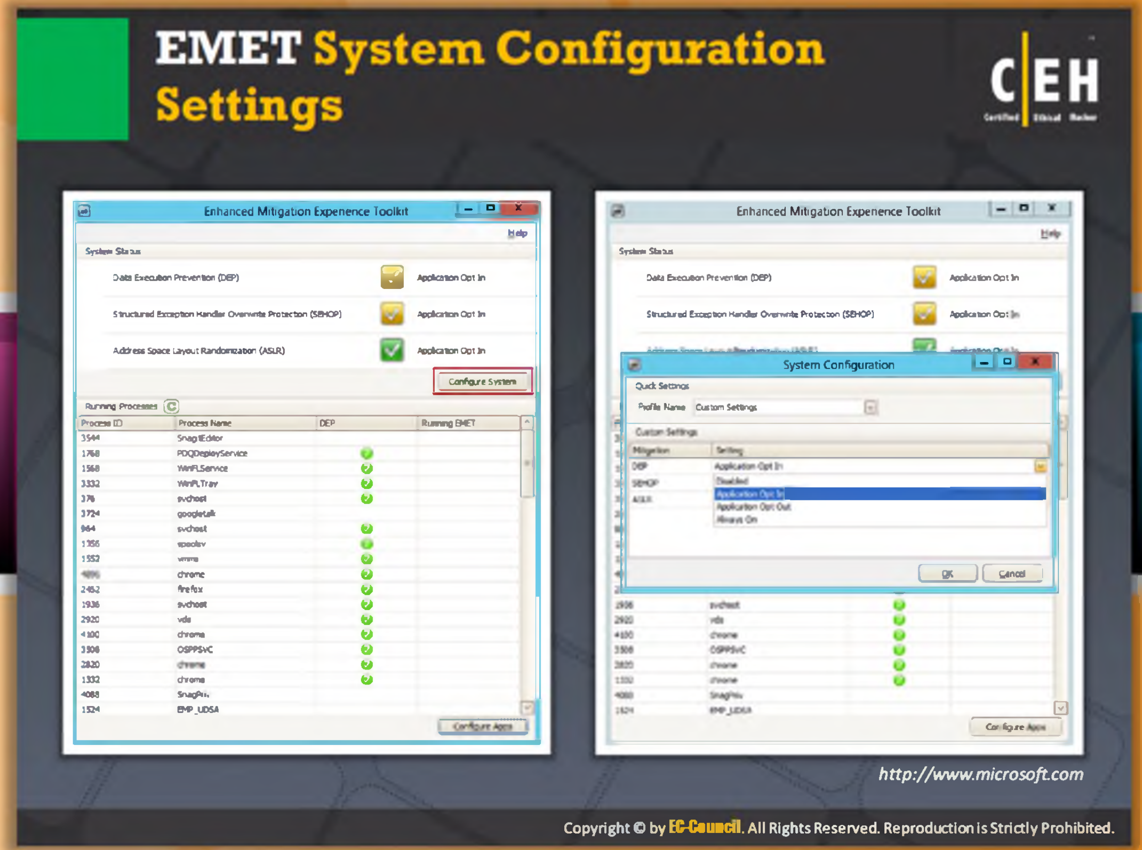
- **Dynamic Data Execution Prevention (DDEP):**

  It marks portions of a process's memory non-executable, making it difficult to exploit **memory corruption vulnerabilities**.

- **Address Space Layout Randomization (ASLR):**

  New in EMET 2.0 is mandatory address space layout randomization (ASLR), as well as non-ASLR-aware modules on all new Windows Versions.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



EMET System Configuration Settings

Source: http://www.microsoft.com

The Enhanced Mitigation Experience Toolkit (EMET) was designed specifically for **preventing vulnerabilities** in software from being **exploited**. After installation, you need to configure EMET to provide protection for software. System and application are the two main **categories** you need to configure on EMET. To configure both these categories, you need to click the respective button present on the right side of the EMET main window. The system status display may vary from one operating system to the other.

The **System Configuration** section is used to configure system-wide (i.e., no need to explicitly define the process to be protected) **specific mitigations** such as DEP, SEHOP, and ASLR. In operating systems such as Windows 7, when the system configuration is set to maximum security, the DEP option will be set to Always On, **SEHOP** to Application Opt Out, and ASLR to Application Opt In modes. But, setting DEP to Always On may cause all applications that are not compatible with DEP to crash. This in turn may cause system instability. Hence, if you wish to accomplish stability, then it is recommended to set all these settings to Application Opt In.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

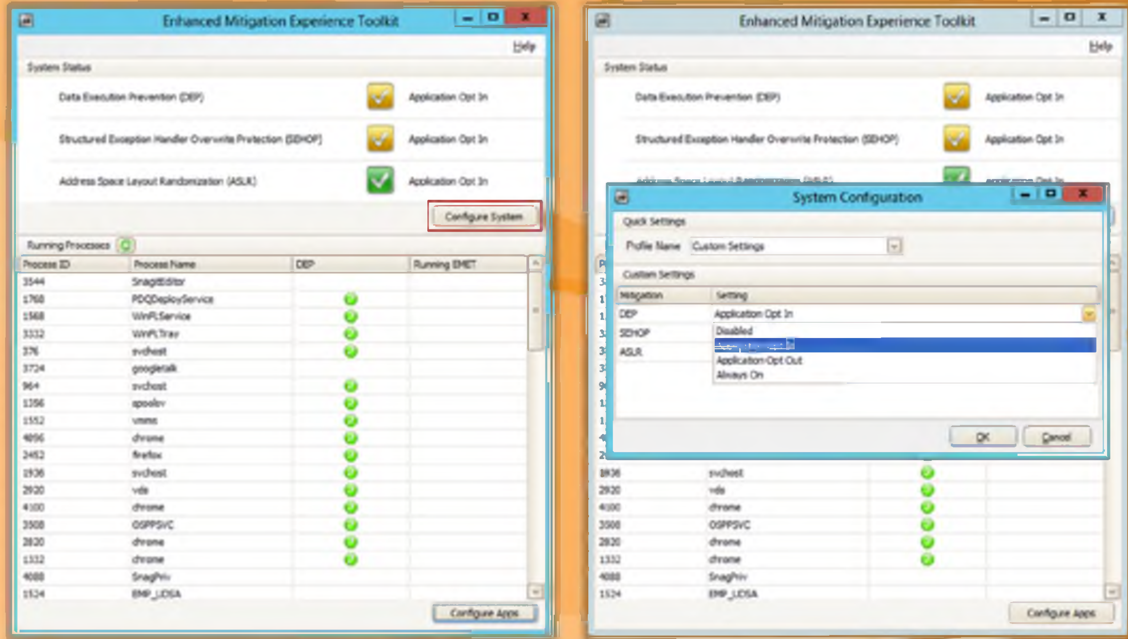
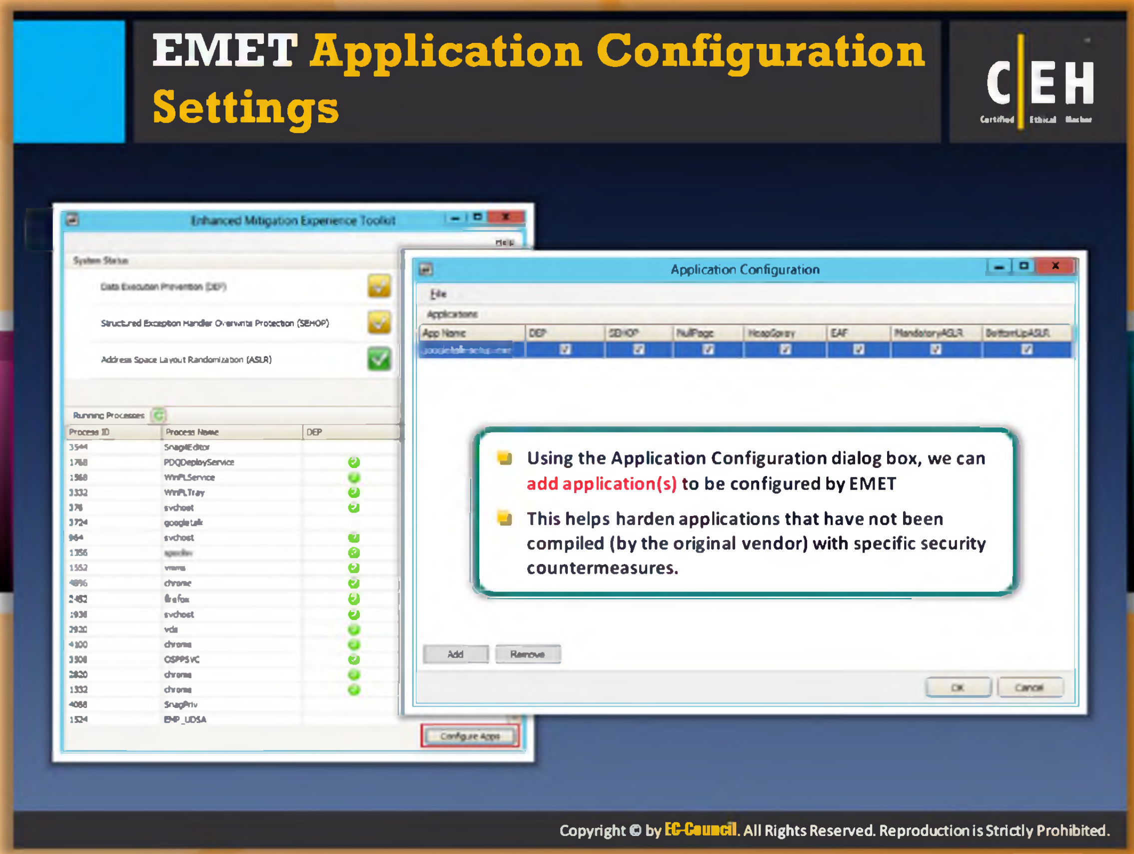


FIGURE 18.14: EMET System Configuration Settings

Ethical Hacking and Countermeasures Copyright © by **EC-Council**
All Rights Reserved. Reproduction is Strictly Prohibited.



EMET Application Configuration Settings

Contrary to system configuration, application configuration enables **mitigations** such as DEP per application rather than **system-wide**. In order to configure applications, you need to click the **Configure Apps** button in EMET's main window. This will prompt you with the Application Configuration window of EMET. By default, this window will be blank. If you want to protect any particular program, then click the Add button and specify the path where the executables of the programs are installed.

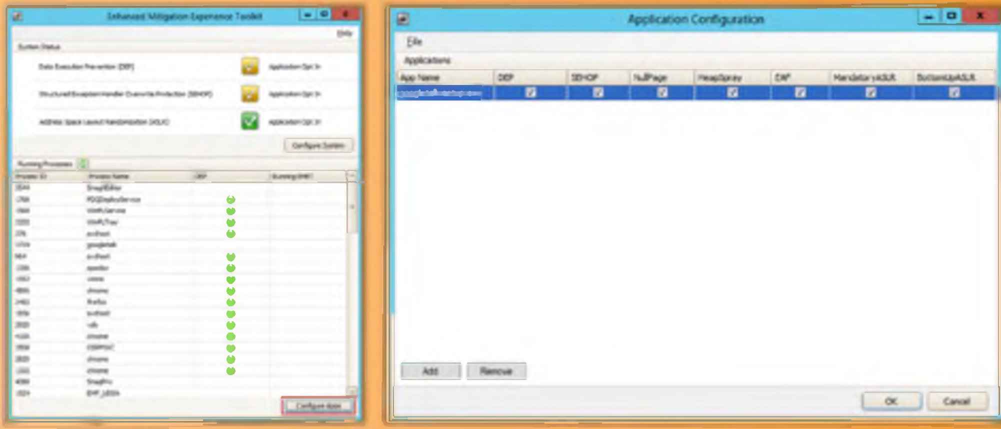
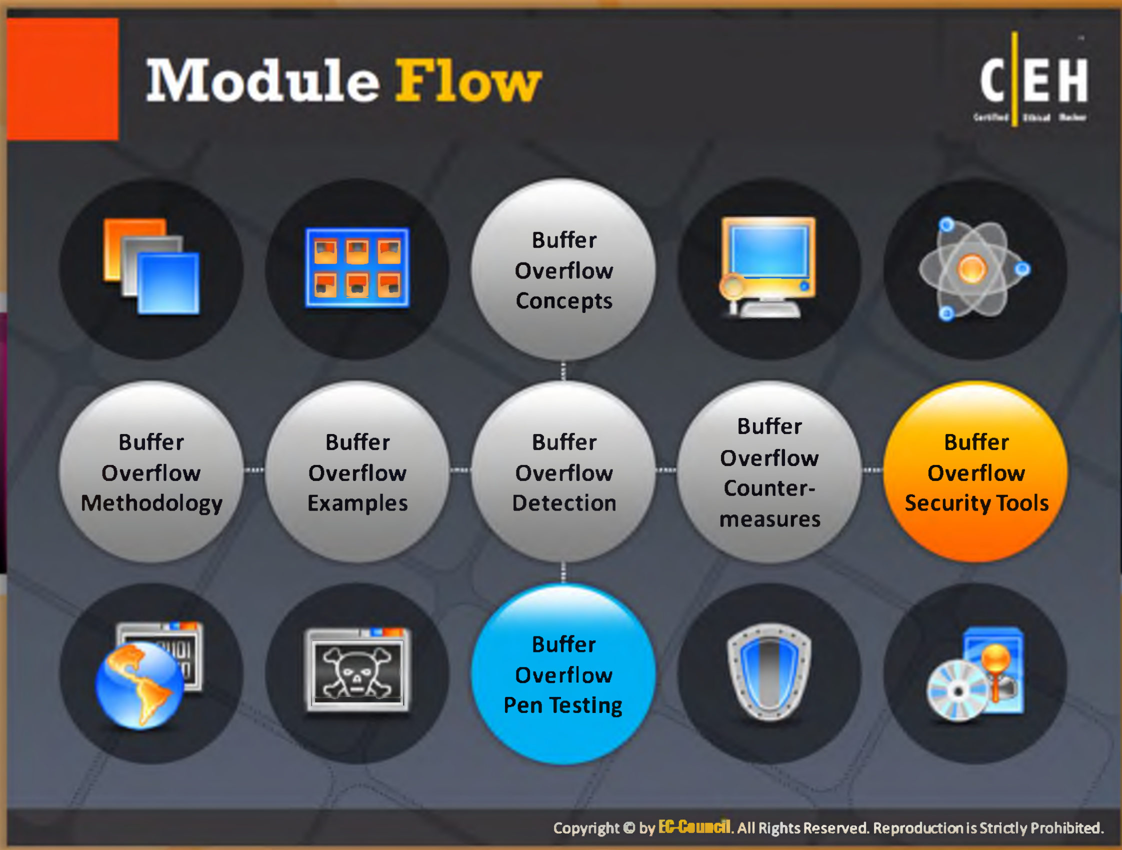


FIGURE 18.15: EMET Application Configuration Settings

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



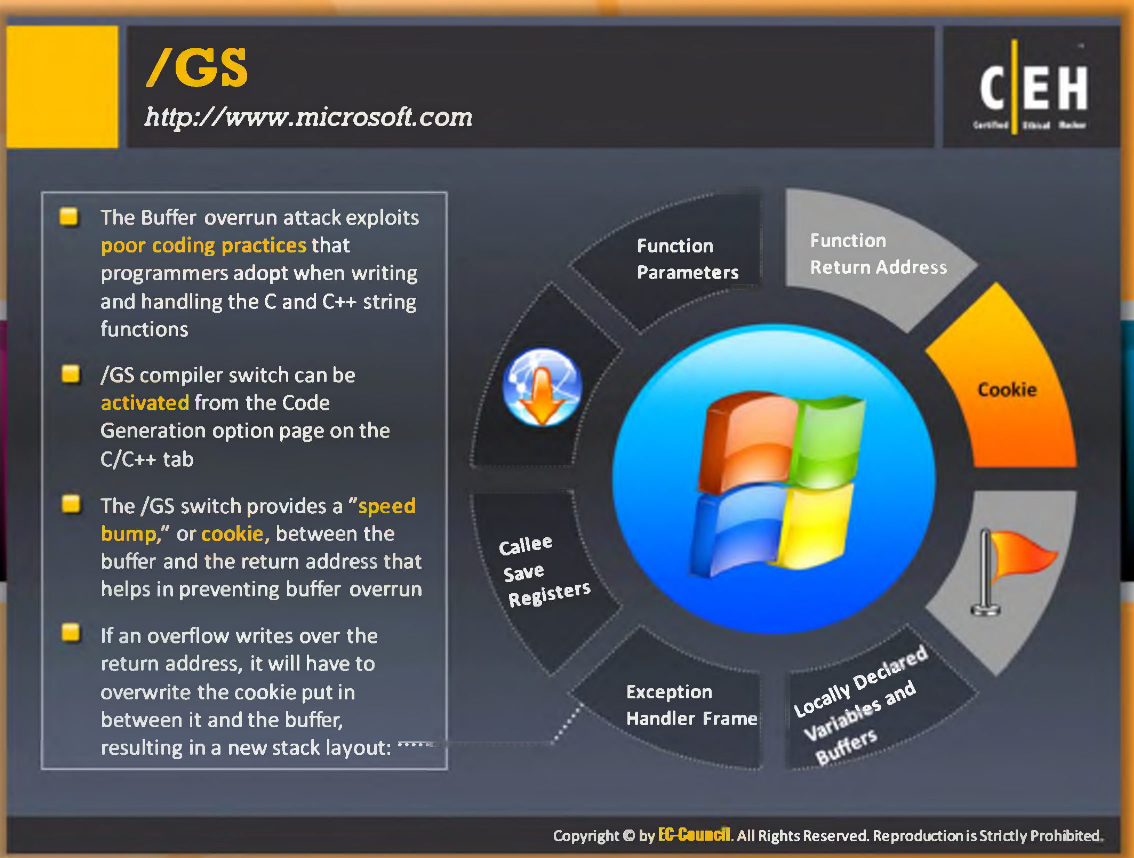
Module Flow

So far, we have discussed what buffer overflow is, how to exploit it, buffer overflow examples, detection methods, and countermeasures. In addition to countermeasures, there are some automated buffer overflow security tools that detect and prevent the exploitation of buffer overflows.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	**Buffer Overflow Security Tools**
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

This section lists and describes buffer overflow security tools.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



/GS

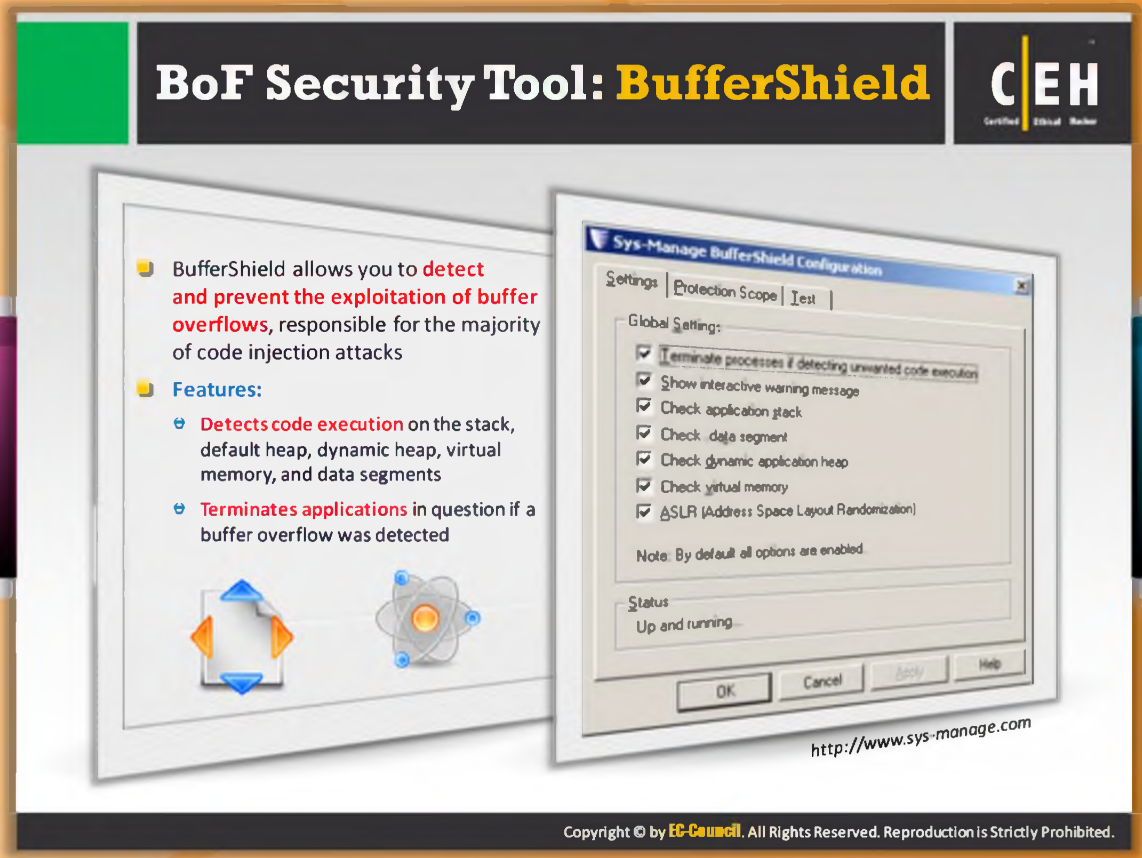
Source: http://www.microsoft.com

The buffer overrun attack utilizes poor coding practices that programmers adopt when writing and handling the C and C++ string functions. The /GS compiler switch can be activated from the Code Generation option page on the C/C++ tab. The /GS switch provides a "speed bump," or cookie, between the buffer and the return address that helps in preventing buffer overrun.

If an overflow writes over the return address, it will have to overwrite the cookie put in between it and the buffer, resulting in a new stack layout:

- Function parameters
- Function return address
- Frame pointer
- Cookie
- Exception Handler frame
- Locally declared variables and buffers
- Callee save registers

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



BoF Security Tool: BufferShield

Source: http://www.sys-manage.com

BufferShield a security tool that prevents the exploitation of buffer overflows. It allows you to detect and prevent the exploitation of **buffer overflows**, responsible for the majority of security related problems.

Key features of BufferShield:

- It detects code execution on the stack, default heap, dynamic heap, virtual memory, and data segments
- It can **terminate** applications in question if a buffer overflow was detected
- It reports to the Windows event log in case of any **detected overflows**
- It allows the definition of a **protection scope** to either protect only defined applications or to exclude certain applications or memory ranges from being protected
- It utilizes Intel XD / AMD NX **hardware based technology** if available
- It has SMP support

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

- It uses Address Space Layout Randomization (ASLR)

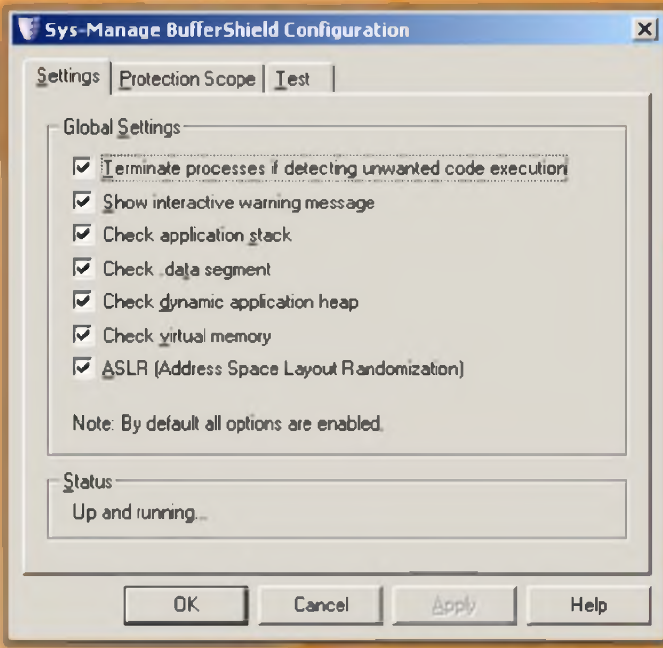
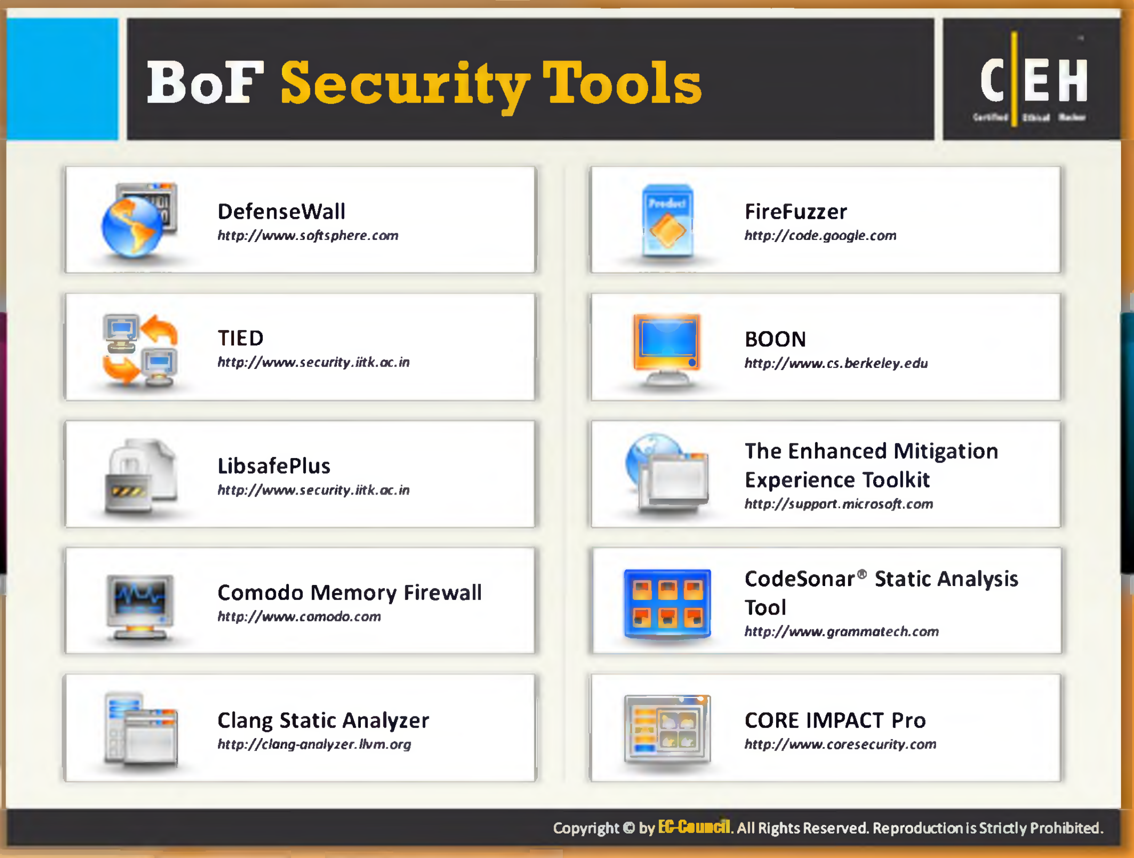


FIGURE 18.16: BufferShield Screenshot

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

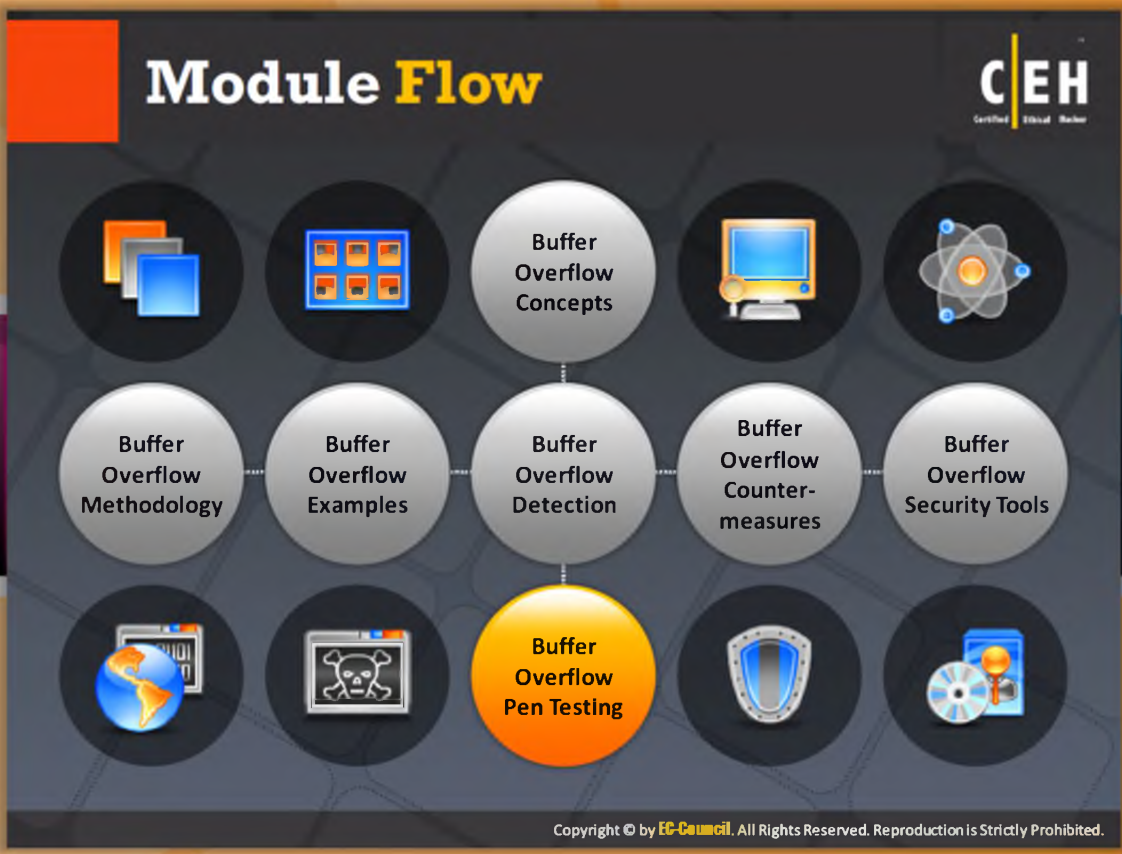


BoF Security Tools

In addition to /GS and BufferSheild, many other buffer overflow security tools are readily available in the market. A more buffer overflow security tools that can detect and prevent buffer overflows are listed as follows:

- DefenseWall availble at http://www.softsphere.com
- TIED available at http://www.security.iitk.ac.in
- LibsafePlus available at http://www.security.iitk.ac.in
- Comodo Memory Firewall available at http://www.comodo.com
- Clang Static Analyzer available at http://clang-analyzer.llvm.org
- FireFuzzer available at http://code.google.com
- BOON available at http://www.cs.berkeley.edu
- The Enhanced Mitigation Experience Toolkit available at http://support.microsoft.com
- CodeSonar® Static Analysis Tool available at http://www.grammatech.com
- CORE IMPACT Pro available at http://www.coresecurity.com

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Module Flow

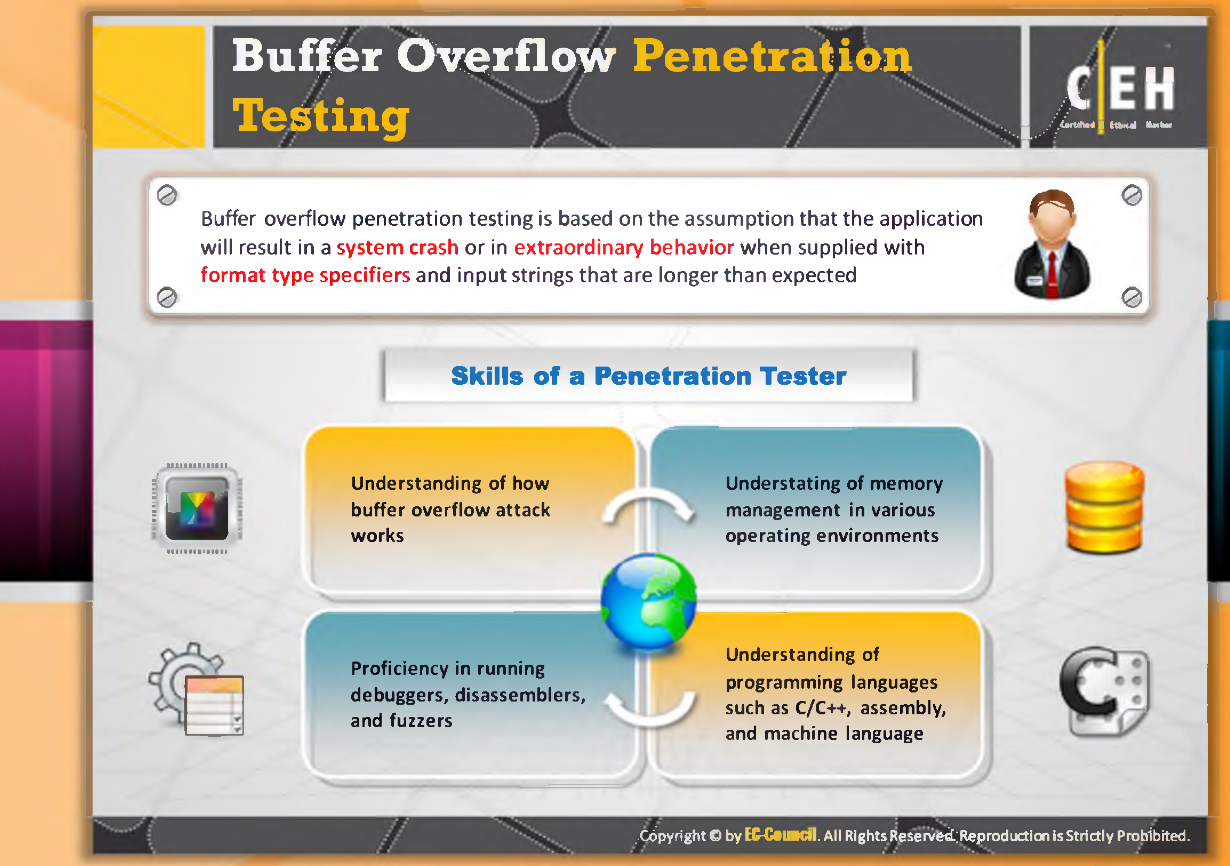
So far, we have discussed all the necessary elements required to test the security of an application or program against buffer overflow vulnerabilities. Now it's time to test the security of an application, service, or program. The test conducted to check the security of your own application by simulating the actions of an attacker or external user is called penetration testing.

Buffer Overflow Concepts	Buffer Overflow Countermeasures
Buffer Overflow Methodology	Buffer Overflow Security Tools
Buffer Overflow Examples	Buffer Overflow Pen Testing
Buffer Overflow Detection	

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

This section provides a detailed step-by-step process of testing the security of application, service, or program against buffer overflow flaws.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



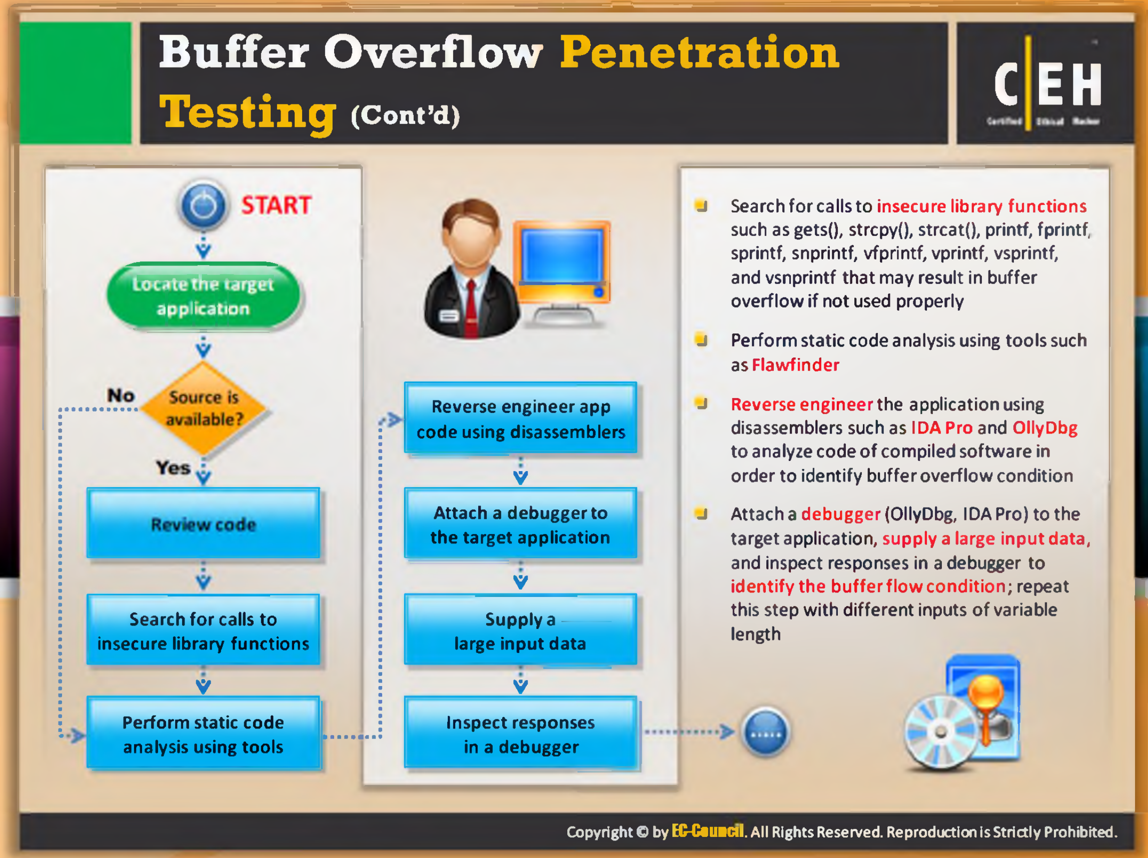
Buffer Overflow Penetration Testing

Buffer overflow penetration testing is based on the assumption that the application will result in a system crash or an extraordinary behavior when supplied with format type specifies and input strings that are longer than expected. A penetration tester's job is to not only scan for the vulnerabilities in the applications or server, but also he need to exploit them to gain access to the remote server.

A good pen tester should possess the following skills:

- Understanding of how buffer overflow attack works
- Understating of memory management in various operating environments
- Understanding of programming languages such as C/C++, assembly, and machine language
- Proficiency in running debuggers, disassemblers, and fuzzers

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Buffer Overflow Penetration Testing (Cont'd)

An application can be tested against buffer overflows by supplying a **larger amount of input data** than the usual. Then you should observe the **application's responses** and **execution** flow to check whether an overflow occurred or not. To test the application against buffer overflow with all possible cases, follow these steps:

Step 1: Locate the target application

In order to perform **penetration testing**, first you should locate the target application on which you want to conduct the test. Then check whether the source code of the **target application** is available or not.

If the source is not available, then go to step 4 to perform **static code analysis** using tools and if the source is available, then review the code.

Step 2: Review code

Review the source code of the application to find the vulnerabilities in the application development and try to **exploit** those vulnerabilities. Test for common vulnerabilities such as buffer overflows, format string exploits, etc.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Step 3: Search for calls to insecure library functions

Library functions make the application development easy but all the library function calls are not secure. Though they seem to be normal, they can be exploited. Hence, you should search for insecure library function calls and **secure** them from **exploitation**.

Step 4: Perform static code analysis using tools

Static code analysis allows you to test the application without actually executing the application. This is usually done with the help of **automated tools** such as **RATS** and **Flawfinder**.

Step 5: Reverse engineer app code using disassemblers

Reverse engineer app code involves testing the **assembly code** with help of disassemblers passively. In this method of testing, various sections of the code are scanned for vulnerable **assembly fragment signatures**.

Step 6: Attach a debugger to the target application

In order to locate and validate a **buffer overflow vulnerability**, you need to attach a debugger to the target application.

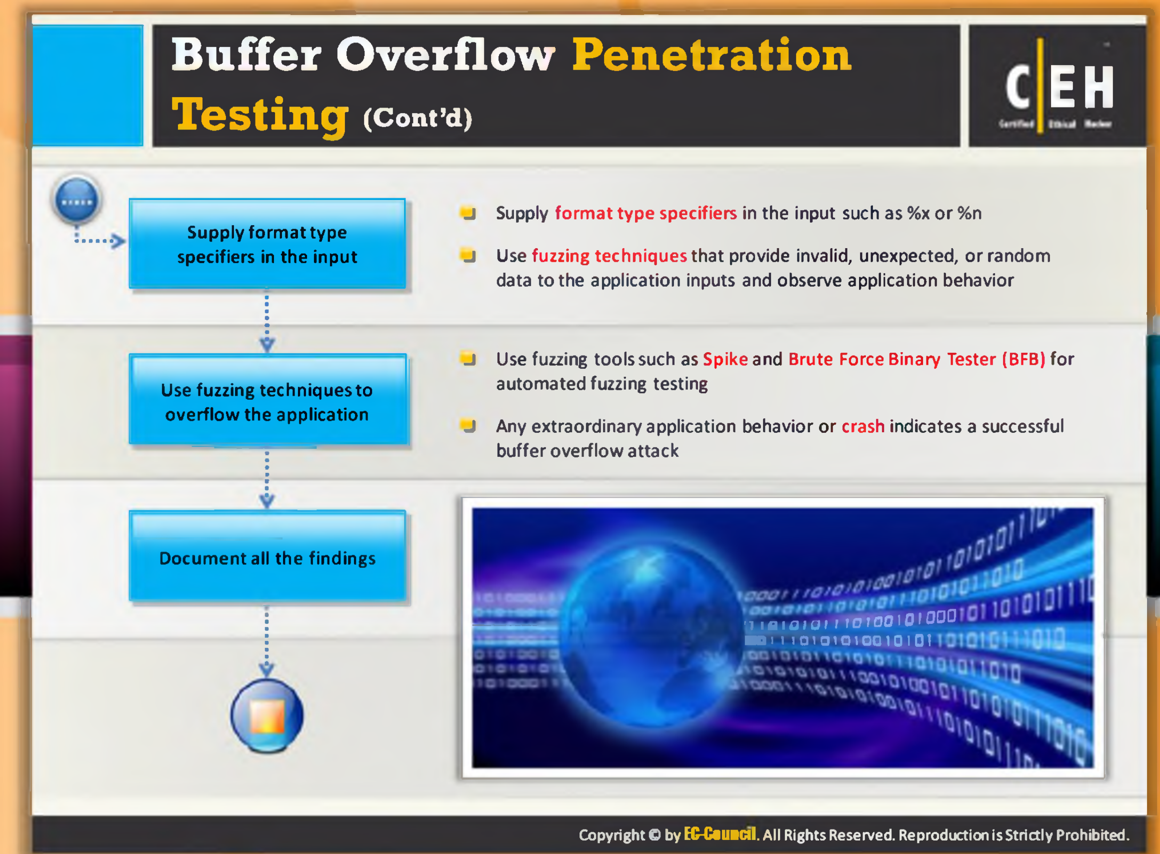
Step 7: Supply a large amount of input data

Create a larger string than the actual size and supply it as input data to the application and observe the responses of the application to the given input.

Step 8: Inspect responses in a debugger

The debugger attached to the application allows you to see the execution flow and state of registers when the buffer overflow gets exploited.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.



Buffer Overflow Penetration Testing (Cont'd)

Step 9: Supply format type specifiers in the input

Supply format type specifiers such as %x or %n in the application input to test for format string vulnerabilities that in turn may lead to buffer overflows.

Step 10: Use fuzzing techniques to overflow the application

Provide invalid, unexpected, or random data as input to the target application using fuzzing techniques and then observe the application behavior. This way you can find whether the application is vulnerable to buffer overflows or not. You can also conduct automated fuzzing test with the help of fuzzing tools such as Spike and Brute Force Binary Tester (BFB).

Step 11: Document all the findings

Documenting all the findings is the last and the most important step that should be carefully carried out. It is the most important step because in this step you need to document all the critical information that can lead to exploitation. Sometimes even a small piece of information left out may lead to great losses for a company. Therefore, this step should be done carefully.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Module Summary

- A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold
- Buffer overflow attacks depend on lack of boundary testing, and a machine that can execute a code that resides in the data or stack segment
- A stack-based buffer overflow occurs when a buffer has been overrun in the stack space
- Buffer overflow vulnerability can be detected by skilled auditing of the code as well as boundary testing
- Shellcode is machine level code used as payload in the exploitation of a software vulnerability
- Countermeasures include checking the code, disabling stack execution, supporting a safer C library, and using safer compiler techniques
- Tools like stackguard, Immunix, and vulnerability scanners help in securing systems

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Module Summary

- A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold.
- Buffer overflow attacks depend on the lack of **boundary testing** and a machine that can execute a code that resides in the data or **stack segment**.
- A stack-based buffer overflow occurs when a buffer has been overrun in the stack space.
- A buffer overflow **vulnerability** can be detected by **skilled auditing** of the code as well as boundary testing.
- Shellcode is small code used as payload in the exploitation of a software vulnerability.
- Countermeasures include checking the code, disabling stack execution, supporting a safer C library, and using **safer compiler techniques**.
- Tools such as stackguard, Immunix, and vulnerability scanners help in **securing systems**.

Ethical Hacking and Countermeasures Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.