Buffer Overflow

Module 17

Engineered by Hackers. Presented by Professionals.

SECURITY NEWS

November 29, 2010 01:00 PM

InformationWeek

Zero Day Bug Bypasses Windows User Account Control

Local buffer overflow vulnerability tricks Microsoft operating systems into granting an attacker system-level user privileges.

Multiple versions of Microsoft Windows are vulnerable to a previously undisclosed, zero-day buffer-overflow vulnerability that would allow an attacker to gain system-level privileges and take control of the PC.

According to security research firm Vupen, "this issue is caused by a buffer overflow error within the 'win32k.sys' driver when processing certain registry values stored as 'reg_binary,' which could allow unprivileged users to crash an affected system or execute arbitrary code with kernel (system) privileges," by modifying registry values related to end-user-defined characters (EUDC) for fonts.

According to security researcher Chester Wisniewski at Sophos, an attacker can use the EUDC-related key "to impersonate the system account, which has nearly unlimited access to all components of the Windows system."

http://www.informationweek.com

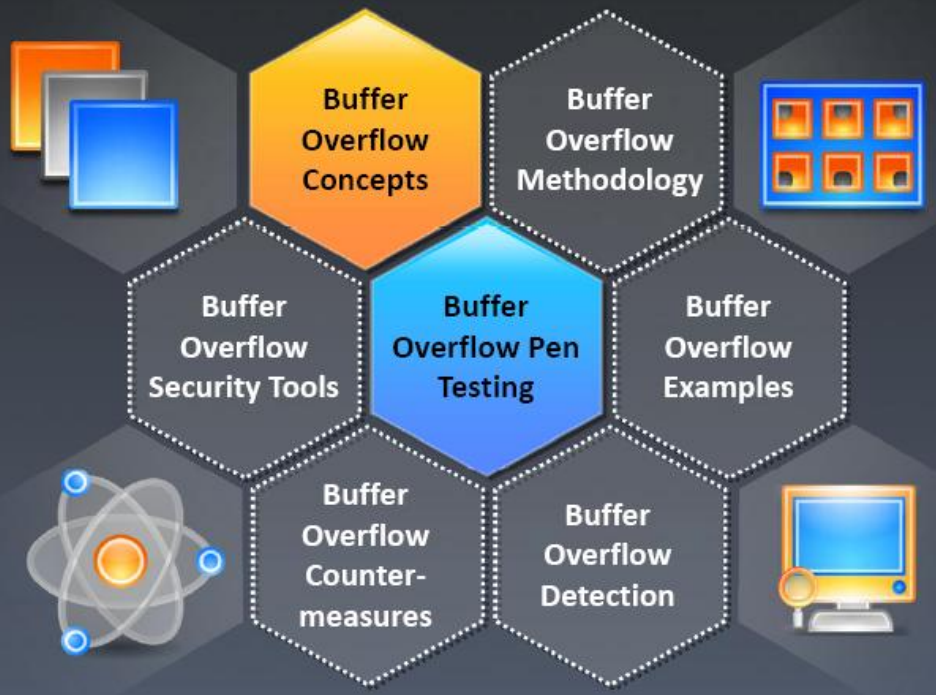
Module Objectives

- Buffer Overflows (BoF)
- Stack-Based Buffer Overflow
- Heap-Based Buffer Overflow
- Stack Operations
- Buffer Overflow Steps
- Attacking a Real Program
- Smashing the Stack
- Examples of Buffer Overflow

- How to Mutate a Buffer Overflow Exploit
- Identifying Buffer Overflows
- Testing for Heap Overflow Conditions: heap.exe
- Steps for Testing for Stack Overflow in OllyDbg Debugger
- BoF Detection Tools
- Defense Against Buffer Overflows
- BoF Countermeasures Tools
- BoF Pen Testing



Module Flow



Buffer Overflows

- A generic buffer overflow occurs when a buffer that has been allocated a specific storage space has more data copied to it than it can handle
- When the following program is compiled and run, it will assign a block of memory 11 bytes long to hold the attacker string
- `strcpy` function will copy the string "DDDDDDDDDDDDDD" into attacker string, which will exceed the buffer size of 11 bytes, resulting in buffer overflow

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

This type of vulnerability is prevalent in UNIX- and NT-based systems

Why are Programs And Applications Vulnerable?

Boundary checks are not done fully or, in most cases, they are skipped entirely

Programming languages, such as C, have vulnerabilities in them

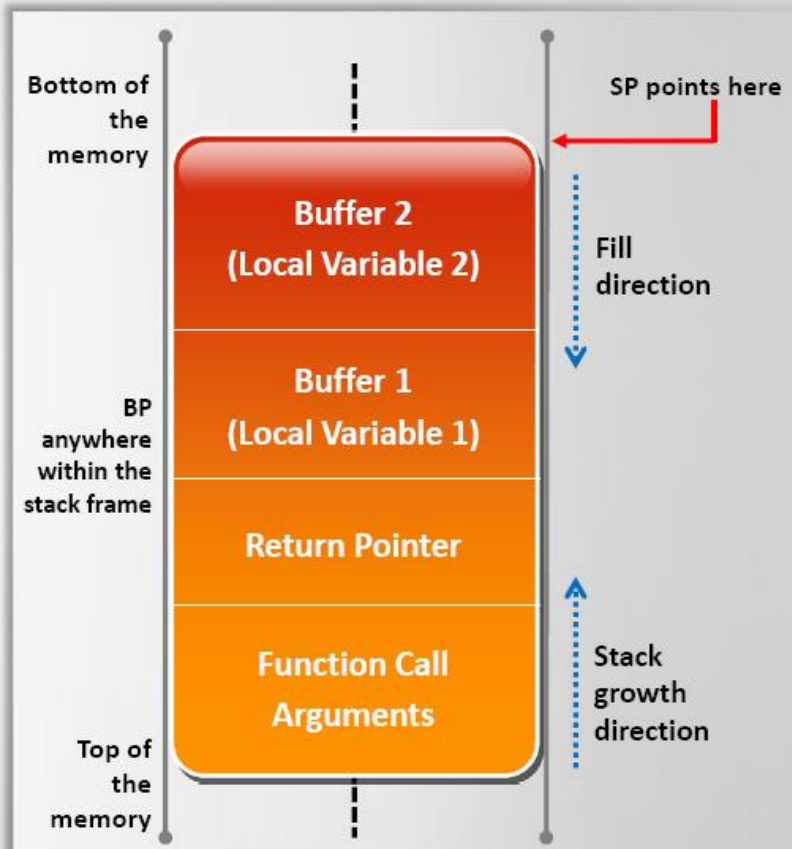
Programs and applications do not adhere to good programming practices

The `strcat()`, `strcpy()`, `sprintf()`, `vsprintf()`, `bcopy()`, `gets()`, and `scanf()` functions in C language can be exploited as they do not check for buffer size

CEH Certified Ethical Hacker

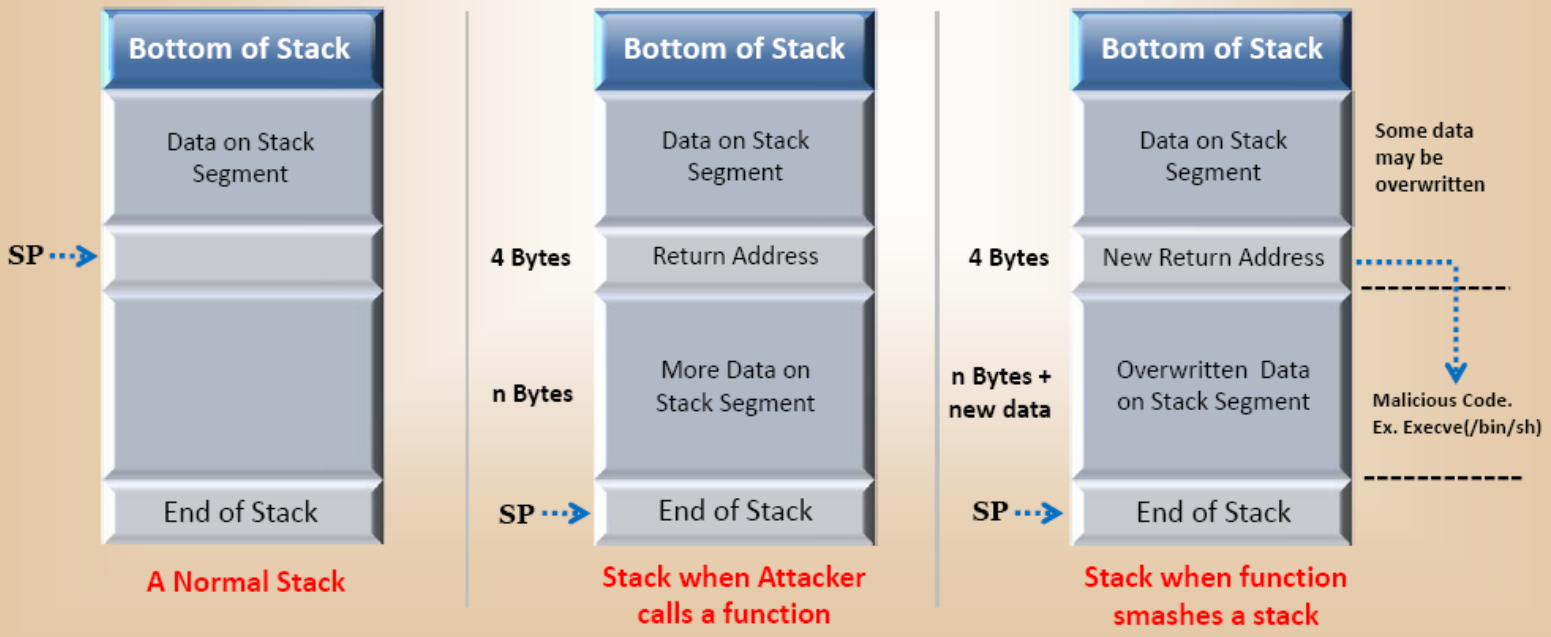
Understanding Stacks

- Stack uses the **Last-In-First-Out (LIFO)** mechanism to pass arguments to functions and refer the local variables
- It acts like a **buffer**, holding all of the information that the function needs
- The stack is created at the **beginning** of a function and released at the **end of it**



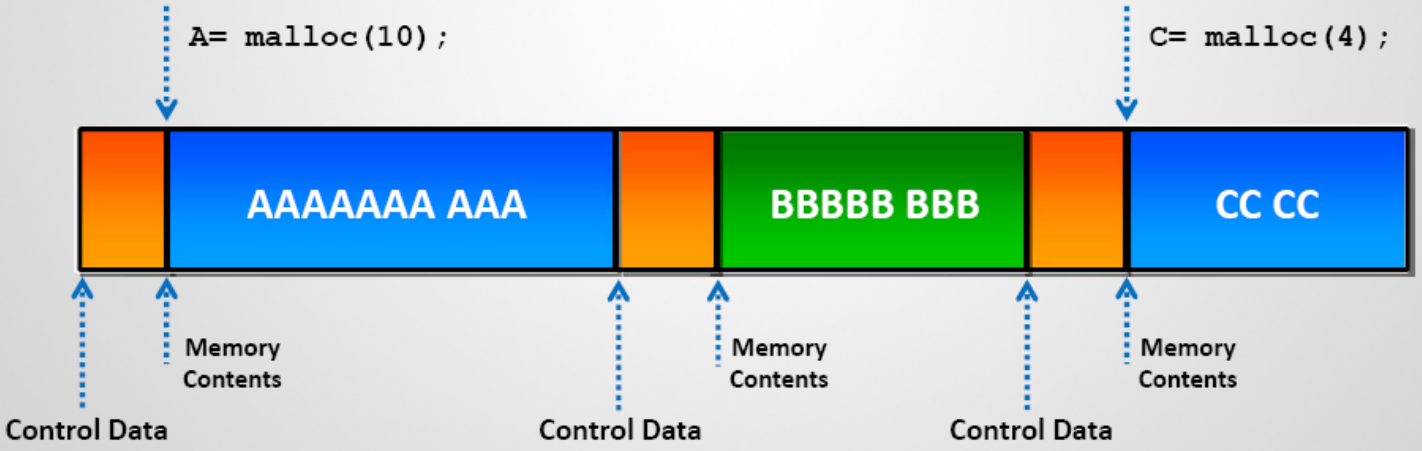
Stack-Based Buffer Overflow

- 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



Understanding Heap

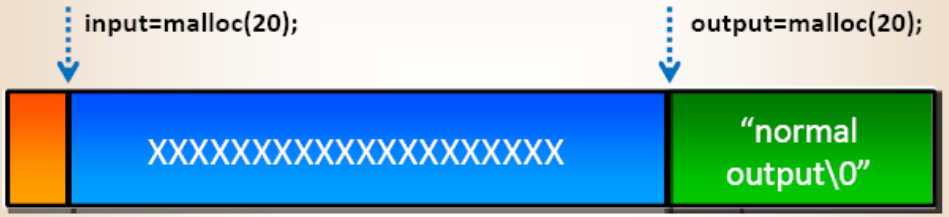
- Heap is an area of memory utilized by an application and is allocated dynamically at the run time with functions, such as `malloc()`
- Static variables are stored on the stack along with the data allocated using the malloc interface
- Heap stores all instances or attributes, constructors, and methods of a class or object



Simple Heap Contents

Heap-Based Buffer Overflow

- 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
- Attacker makes a buffer to overflow on the **lower part of heap**, overwriting other dynamic variables, which can have unexpected and unwanted effects

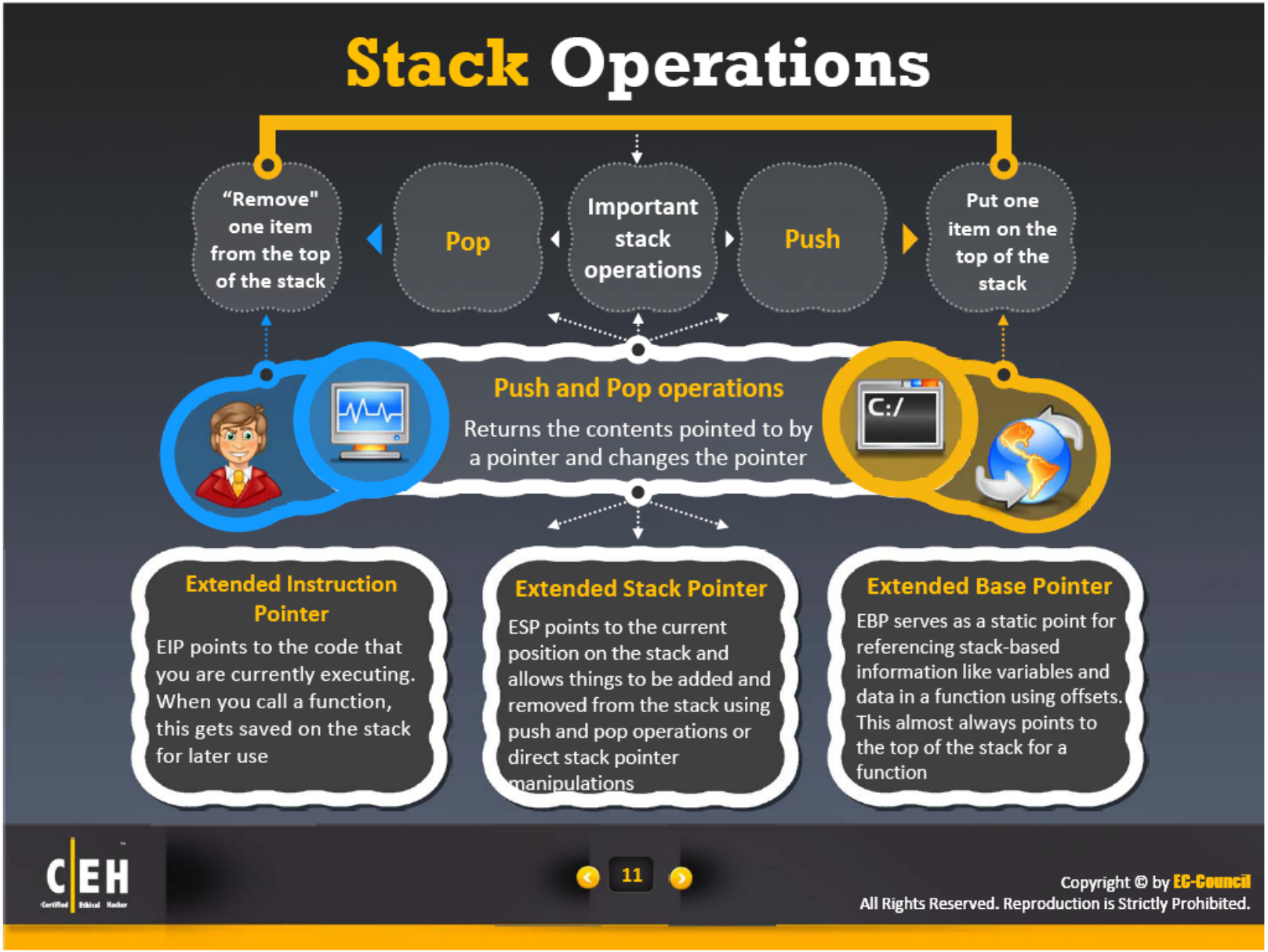


Heap: Before Overflow

Heap: After Overflow

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

Stack Operations



Shellcode

- Shellcode is a small code used as **payload** in the exploitation of a software vulnerability
- 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

Example

```
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
"\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
"\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
"\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01"
```

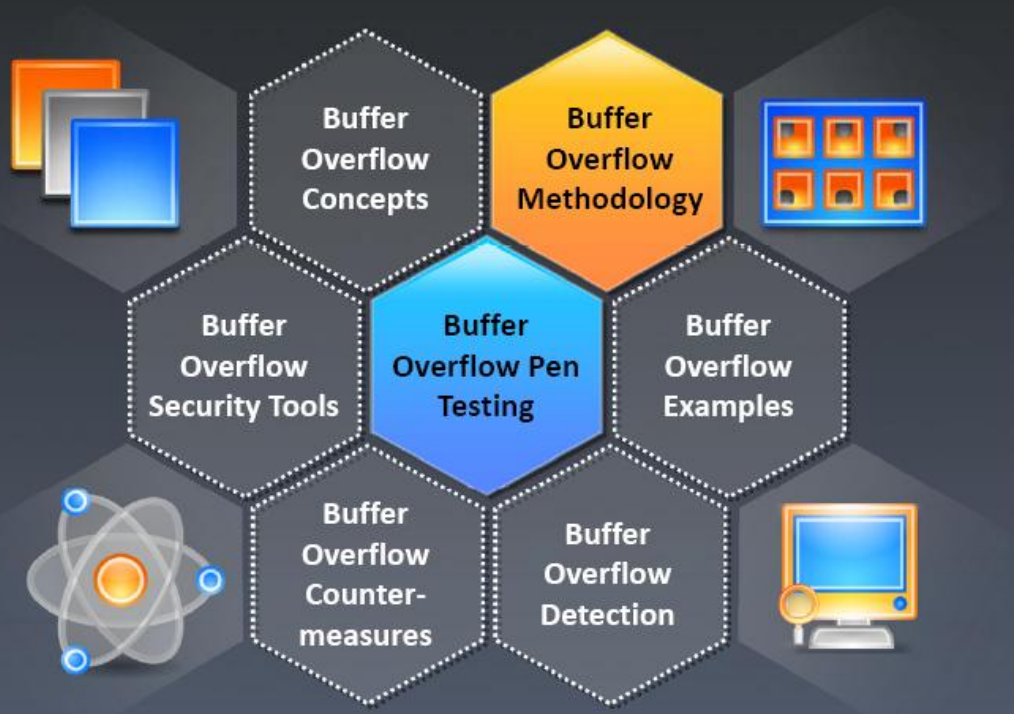
No Operations (NOPs)

- Most CPUs have a No Operation (NOP) instruction – it does nothing but advance the instruction pointer
- Usually, you can put some of these ahead of your program (in the string)

  As long as the new return address points to a NOP, it is OK
- Most intrusion detection systems (IDSs) look for **signatures of NOP sleds**

- Attacker pads the beginning of the intended buffer overflow with a **long run of NOP instructions** (a NOP slide or sled) so the CPU will **do nothing** until it gets to the "main event" (which preceded the "return pointer")
- ADMutate (by K2) accepts a buffer overflow exploit as input and randomly creates a functionally equivalent version (polymorphism)

Module Flow



Knowledge Required to Program Buffer Overflow Exploits

Understanding of stack and heap memory processes

Understanding of how system calls work at the machine code level

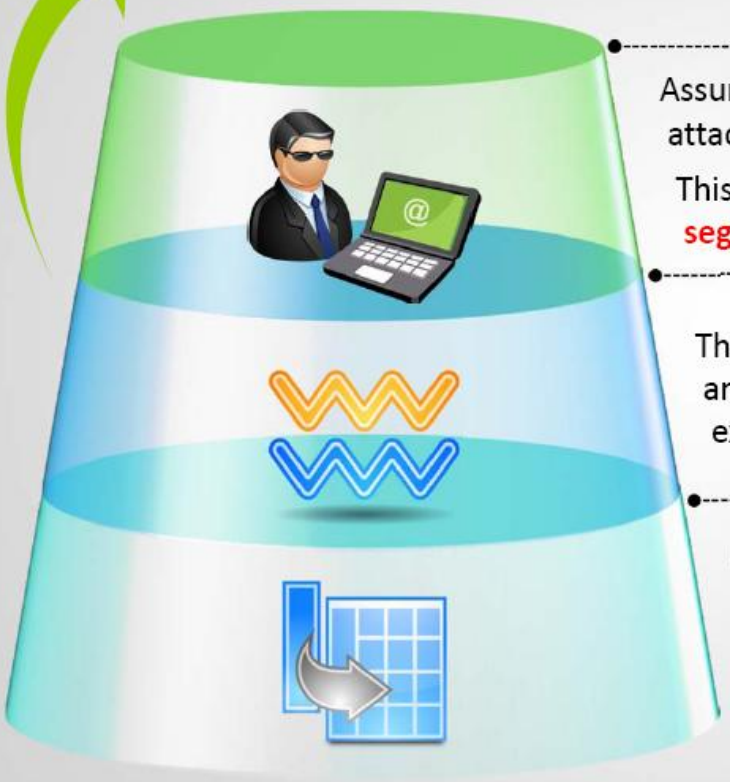
Familiarity with compiling and debugging tools such as gdb

Knowledge of assembly and machine language

Knowledge of C and Perl programming language

Buffer Overflow Steps

Attacking a Real Program



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 user has to insert code in the input, he or she has to know the **exact address** and **size** of the stack and make the return pointer point to his code for execution

Format String Problem

In C, consider this example of Format string problem:

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

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

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

Correct form is:

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

Overflow using Format String

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

```
char errmsg[512],
outbuf[512];
sprintf (errmsg, "Illegal
command: %400s", user);
sprintf( outbuf, errmsg );
```

What if `user = "%500d <nops> <shellcode>"`

- Bypass "%400s" limitation
- Will overflow outbuf

Smashing the Stack

The general idea is to overflow a buffer so that it overwrites the return address

When the function is done it will jump to whatever address is on the stack

Buffer overflow allows us to change the return address of a function

Put some code in the buffer and set the return address to point to it

Once the Stack is Smashed...

Gain Access

- Once the vulnerable process is commandeered, the attacker has the same privileges as the process and can gain normal access
- He or she can then exploit a local buffer overflow vulnerability to gain super-user access

Use Netcat

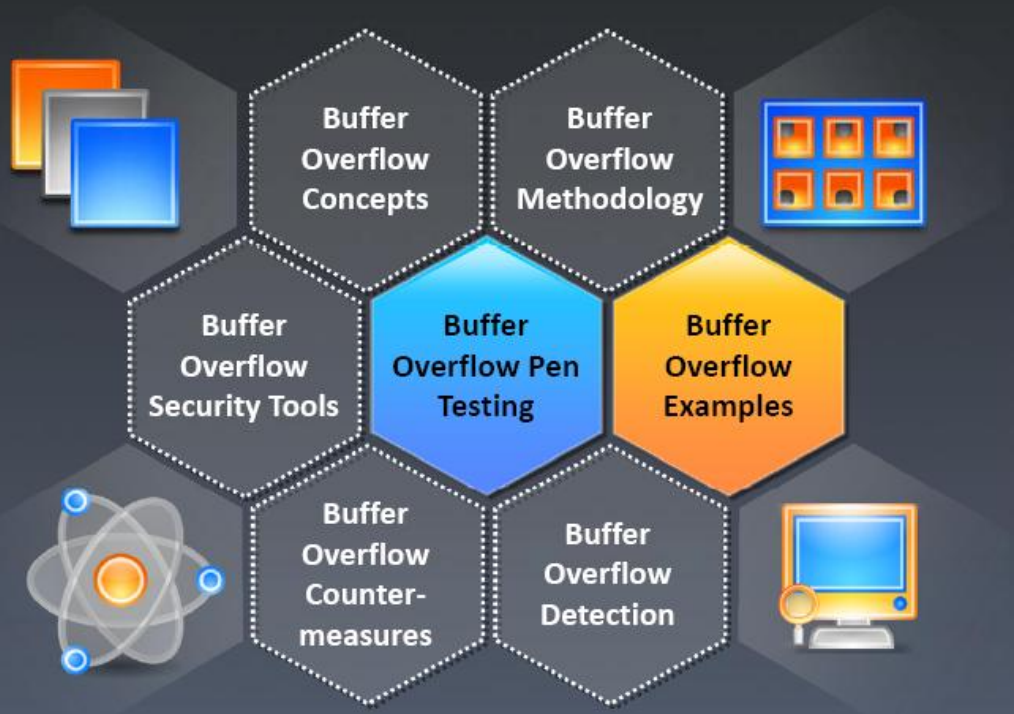
Create a backdoor

- Using (UNIX-specific) inetd
- Using Trivial FTP (TFTP) included with Windows 2000 and some UNIX flavors

Use Netcat to make raw and interactive connections

- UNIX-specific GUI
- Shoot back an Xterminal connection

Module Flow



Simple Uncontrolled Overflow

Example of Uncontrolled Stack Overflow

```
/* 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];
strcpy(buffer,"AAAAAAAAAAAAAAAAAAAA");
/*copy 20 bytes of A into the buffer*/
return 1; /*return, this will cause an access violation due to stack corruption.*/ }
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("Lets Go\n");
return 1; /*leaves the main function*/ }
```

Example of Uncontrolled Heap 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);
}
```

Simple Buffer Overflow in C

Vulnerable C Program overrun.c

```
#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);
}
```

- 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

Code Analysis

- The "**code gets**", 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
```

Exploiting Semantic Comments in C (Annotations)

Adding "@" after the "/*"

- Adding "@" after the "/*" which is considered a comment in C) is recognized as syntactic entities by **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@*/

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

How to Mutate a Buffer Overflow Exploit

For the NOP Portion

- **Randomly replace** the NOPs with functionally equivalent segments of the code (e.g.: x++; x-; ? NOP NOP)

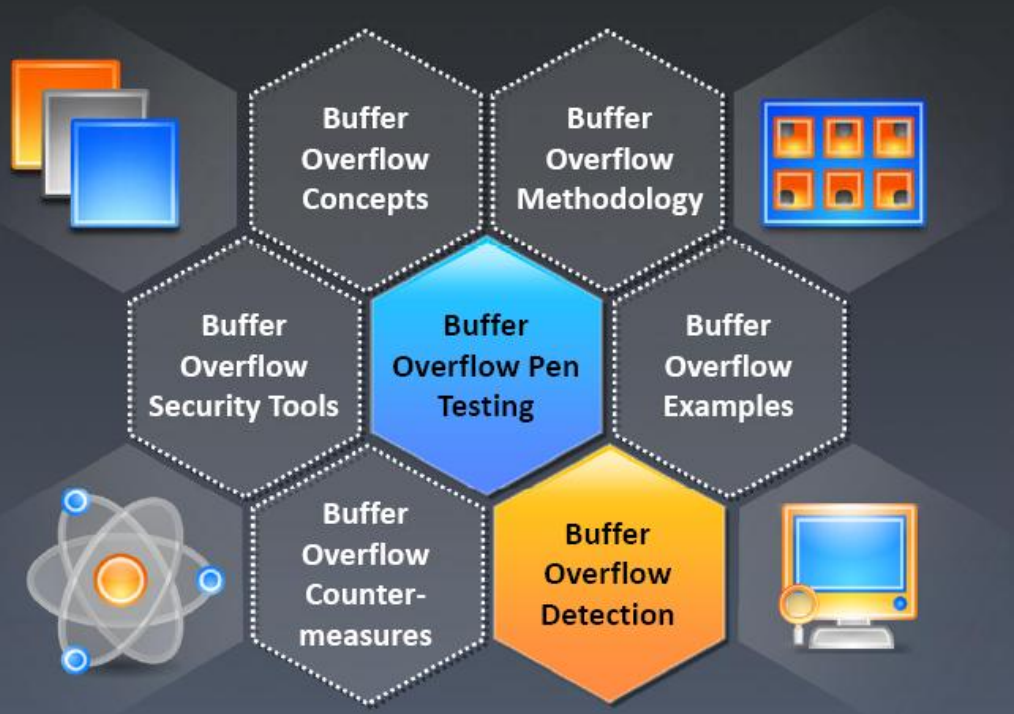
For the "Main Event"

- Apply **XOR** to combine code with a random key unintelligible to IDS. The CPU code must also **decode** the gibberish in time in order to run the decoder. By itself, the decoder is **polymorphic** and therefore hard to spot

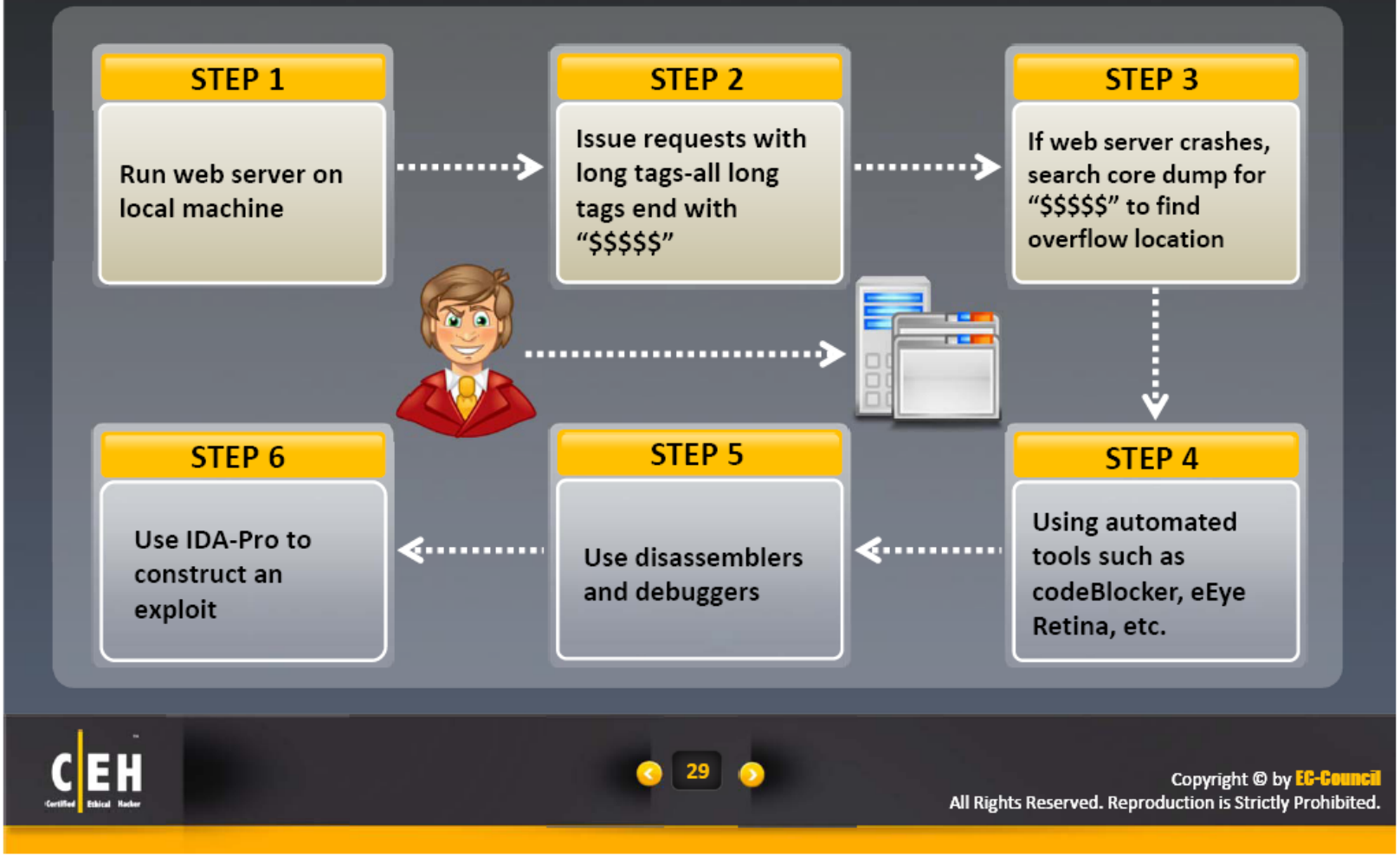
For the "Return Pointer"

- Randomly **tweak LSB** of the pointer to land in the NOP-zone

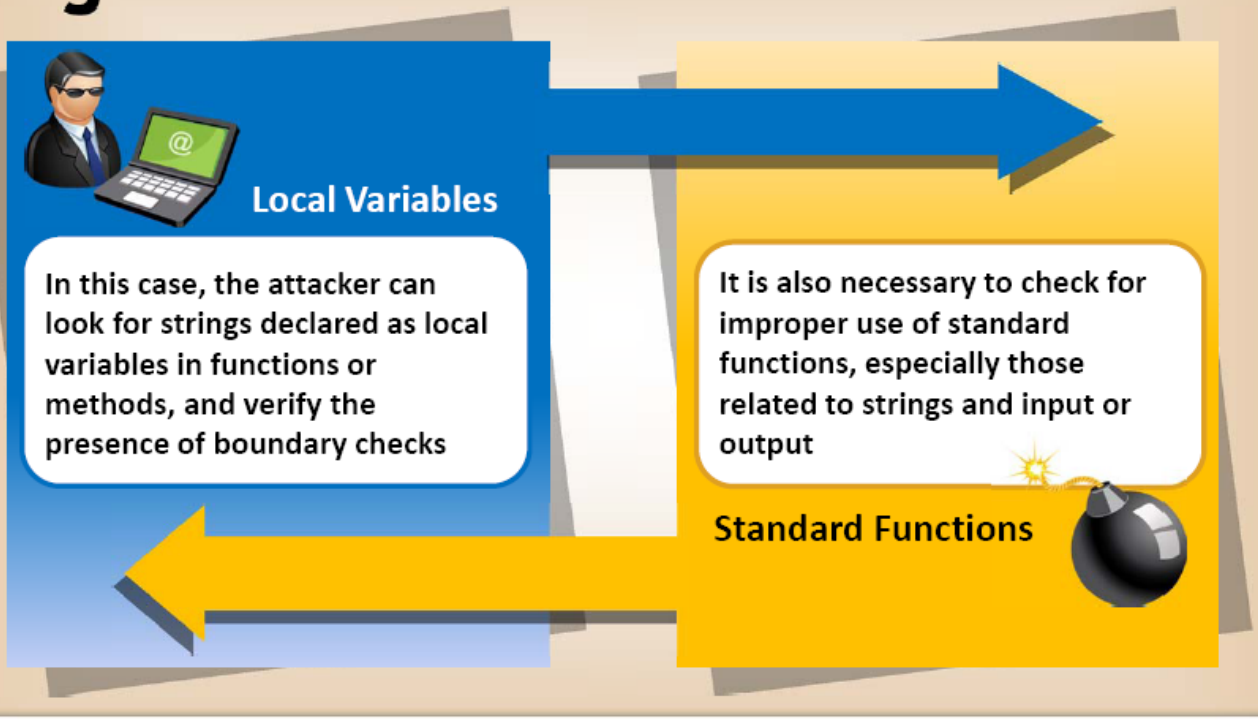
Module Flow



Identifying Buffer Overflows



How to Detect Buffer Overflows in a Program?



Another way is to feed the application with huge amounts of data and check for abnormal behavior

BOU (Buffer Overflow Utility)

- The BOU tool can be used by an attacker to test Web apps for buffer overflow conditions

The tool needs two inputs:

- The "request" file, which is to be tested
- How much of the code should be attacked (specified in a file called "command")

- It takes a request file that is to be tested and outputs all of the activity to STDOUT based on the level of verbosity specified

Example of the 'command' file

```
key=account_number values=12345678900000
times=40
```

Example of the 'request' file

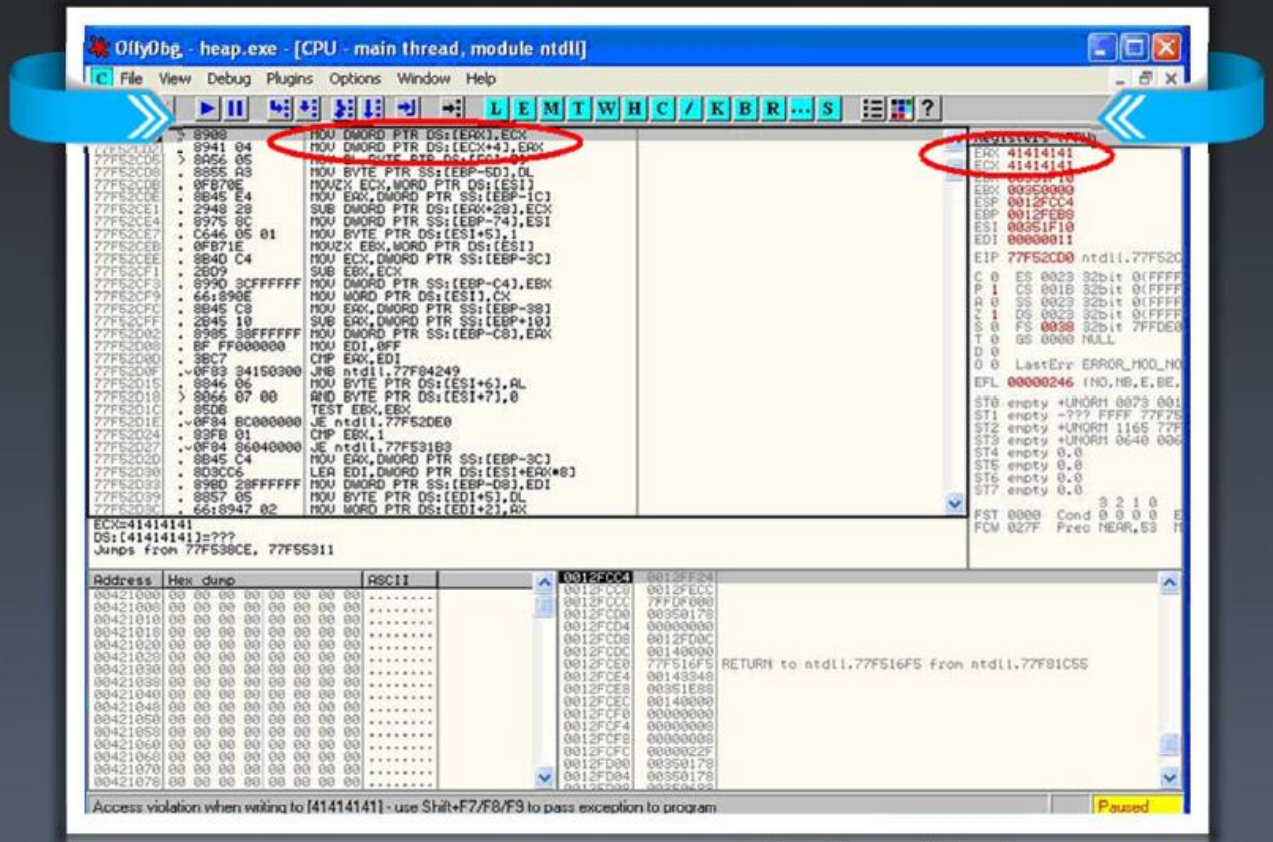
```
POST
http://192.168.1.200:8080/WebGoa
t/attack HTTP/1.0

Referer:
http://192.168.1.200:8080/WebGoa
t/attack

Content-Type: application/x-www-
form-urlencoded

Proxy-Connection: Keep-Alive

User-Agent: Mozilla/4.0
(compatible; MSIE 6.0; Windows
NT 5.1; SV1;)

Host: 192.168.1.200:8080

Content-Length: 18

Cookie:
JSESSIONID=5396FA44D38F8EE14906F
CBAA7680C55

Authorization: Basic
Z3Vlc3Q6Z3Vlc3Q=

account_number=102
```

Testing for Heap Overflow Conditions: heap.exe

Variants of the heap overflow (heap corruption) vulnerability including those that:	Testing for heap overflows by supplying longer input strings than expected	Two registers EAX and ECX, can be populated with user-supplied addresses
1. Allow overwriting function pointers 2. Exploit memory management structures for arbitrary code execution	1. A pointer exchange taking place after the heap management routine comes into action	1. One of the addresses can point to a function pointer which needs to be overwritten, for example UEF (Unhandled Exception filter) 2. The other address can be the address of user supplied code that needs to be executed

When the MOV instructions shown in the left pane of the screenshot are executed, the overwrite takes place. When the function is called, the user-supplied code gets executed

Testing for Heap Overflow Conditions: heap.exe



http://www.ollydbg.de

Steps for Testing for Stack Overflow in OllyDbg Debugger

Attach a debugger to the target application or process

Generate malformed input for the application

Subject the application to malformed input

Inspect responses in a debugger

Testing for Stack Overflow in OllyDbg Debugger

Demonstration of how an attacker can overwrite the instruction pointer (with user-supplied values) and control program execution

Step 1

Testing "sample.exe" 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;
}
```

Step 2

Launch "sample.exe" in a debugger

Step 3

A large sequence of characters such as "A", can be supplied in the argument field as shown

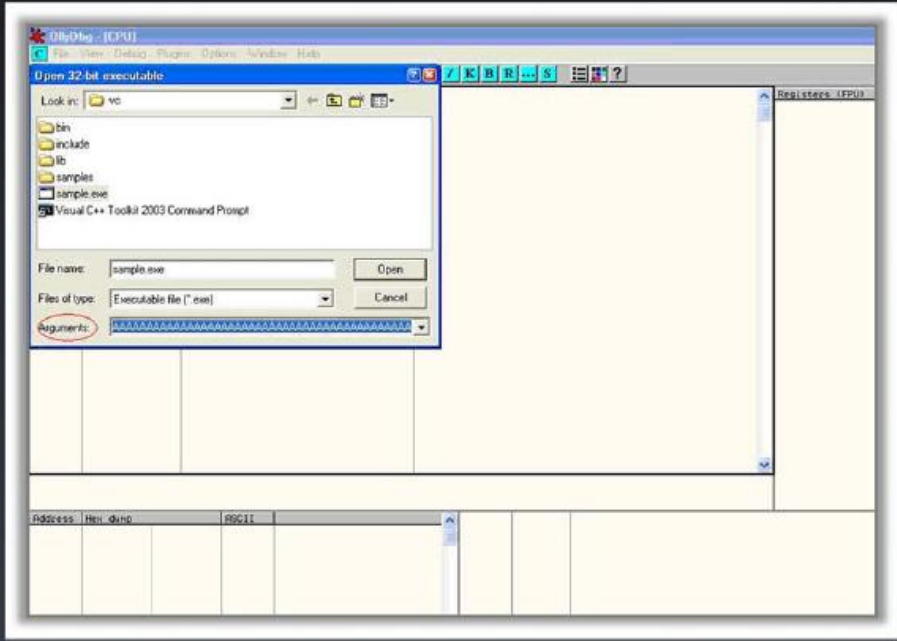
Step 4

Open the executable with the supplied arguments (AAAAAAAA...) and continue execution, result is shown in fig

Step 5

EIP contains the value "41414141", which represents the hexadecimal "AAAA"

Testing for Stack Overflow in OllyDbg Debugger



Testing for Format String Conditions using IDA Pro

Format String Vulnerabilities

- Format string vulnerabilities manifest mainly in:
  - Web servers
  - Application servers
  - Web applications utilizing C/C++ based code
  - CGI scripts written in C

Manipulating Input Parameters

- Attacker manipulates input parameters to include %x or %n type specifies

  For example a legitimate request like:

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

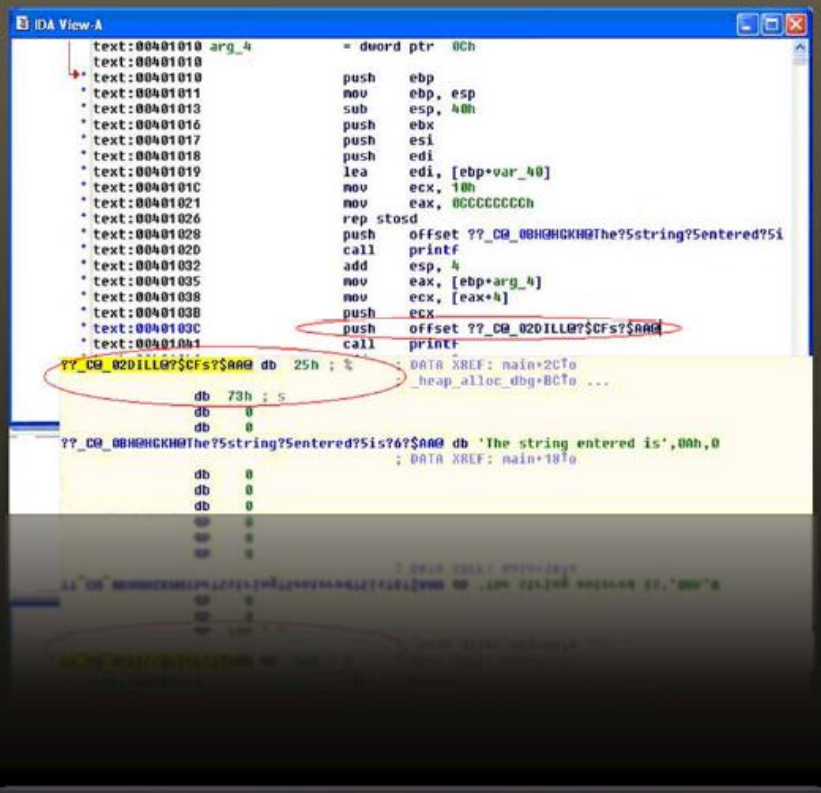
  to

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

Testing for Format String Conditions using IDA Pro

- Attacker identifies the presence of a format string vulnerability by checking instances of code (assembly fragments)
- When the disassembly is examined using IDA Pro:
  - The address of a format type specifier being pushed on the stack is clearly visible before a call to printf is made

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



BoF Detection Tools

BOU (Buffer Overflow Utility)
http://www.net-security.org

Flawfinder
http://www.dwheeler.com

OllyDbg
http://www.ollydbg.de

RATS (Rough Auditing Tool for Security)
https://www.fortify.com

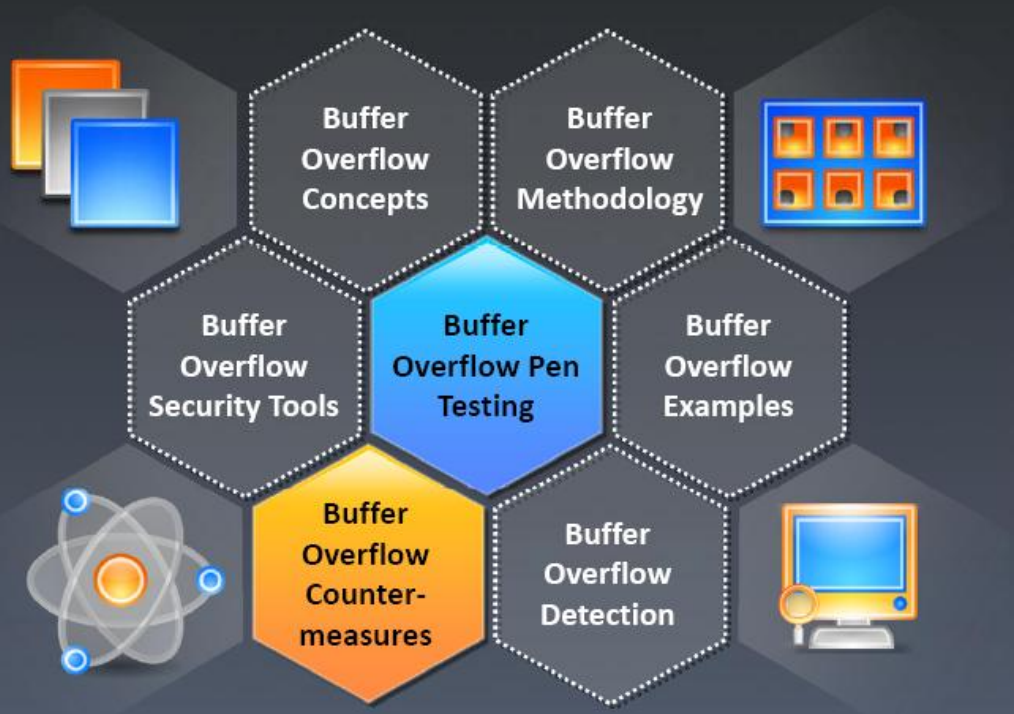
Splint
http://www.splint.org

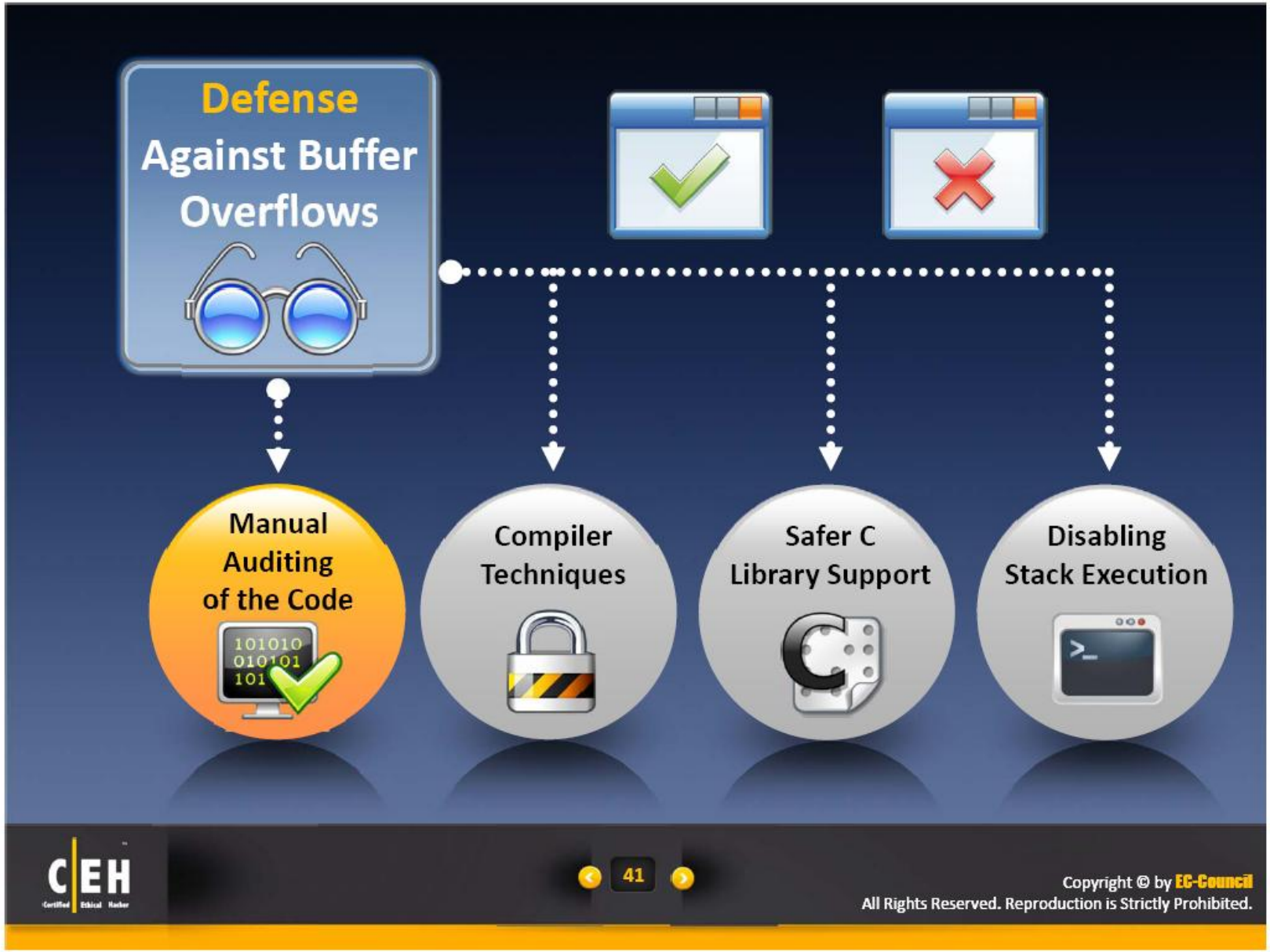
BLAST (Berkeley Lazy Abstraction Software Verification Tool)
http://mtc.epfl.ch

BOON
http://www.cs.berkeley.edu

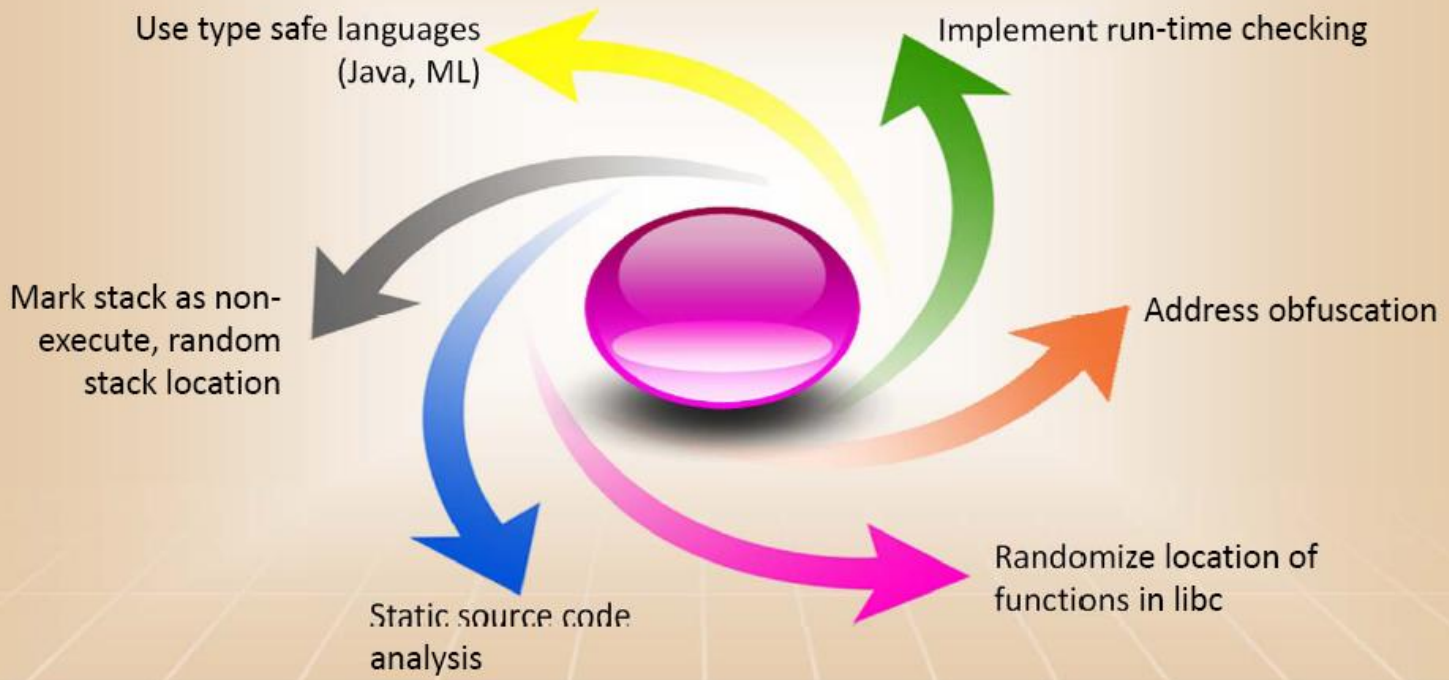
Stack Shield
http://www.angelfire.com

Module Flow

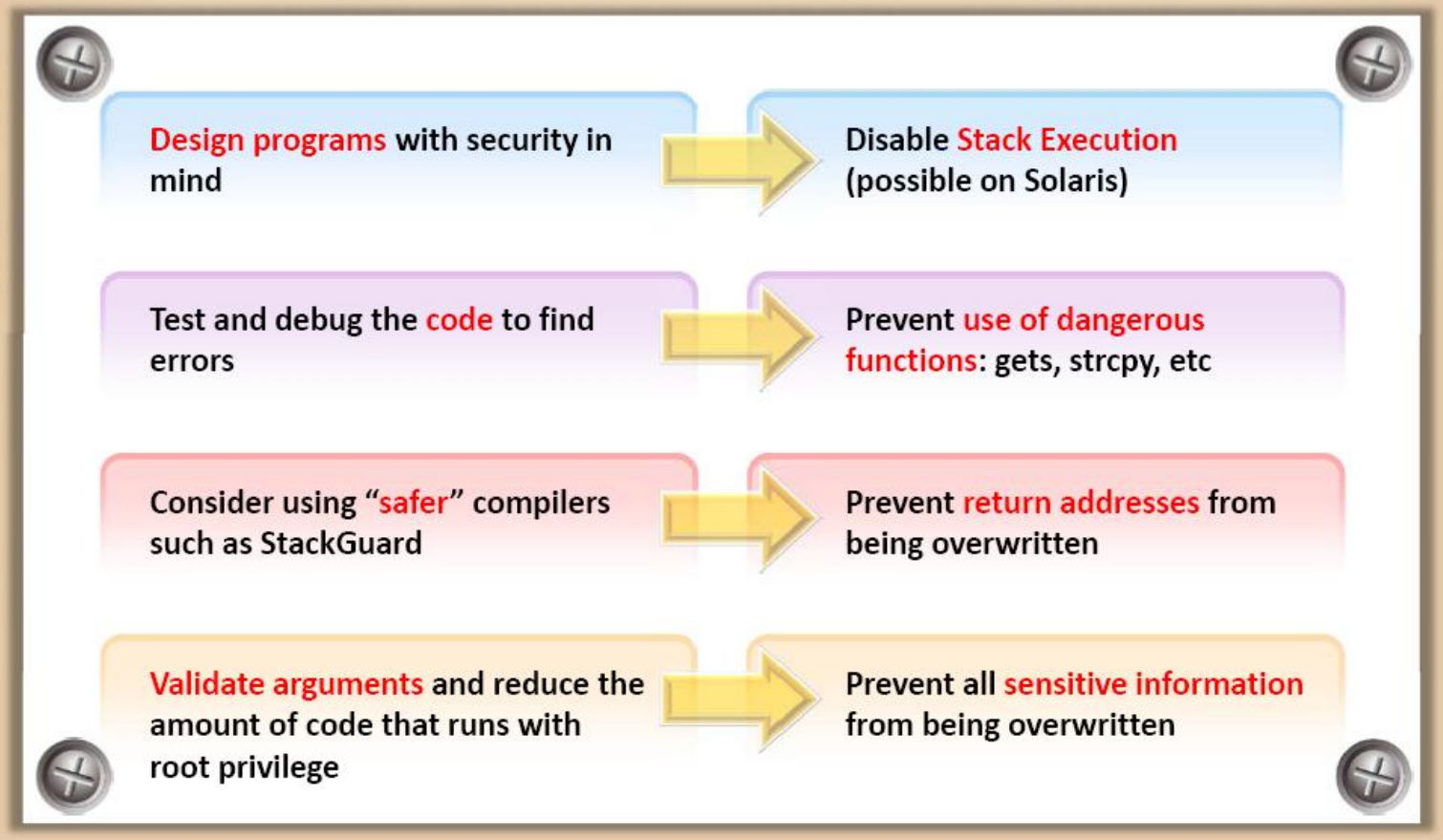



Defense Against Buffer Overflows
Manual Auditing of the Code
101010 010101 101
Compiler Techniques
Safer C Library Support
Disabling Stack Execution
CEH
41
Copyright © by EC-Council
All Rights Reserved. Reproduction is Strictly Prohibited.

Preventing BoF Attacks



Programming Countermeasures



Programming Countermeasures

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

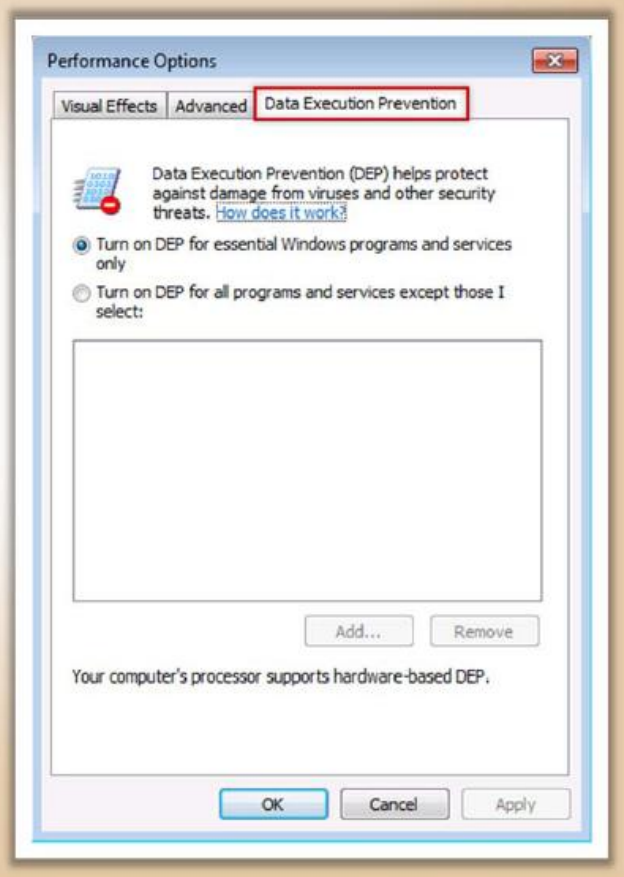
Change the rules at the **operating system level** for which memory pages are allowed to hold executable data

Make use of **safe libraries**

Make use of tools that can **detect buffer overflow vulnerabilities**

Data Execution Prevention (DEP)

- DEP is a set of **hardware** and **software** technologies that monitors programs to verify whether they are using system memory **safe** and **secure**
- It prevent 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 an advantage of exception-handling mechanisms in Windows helps by **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

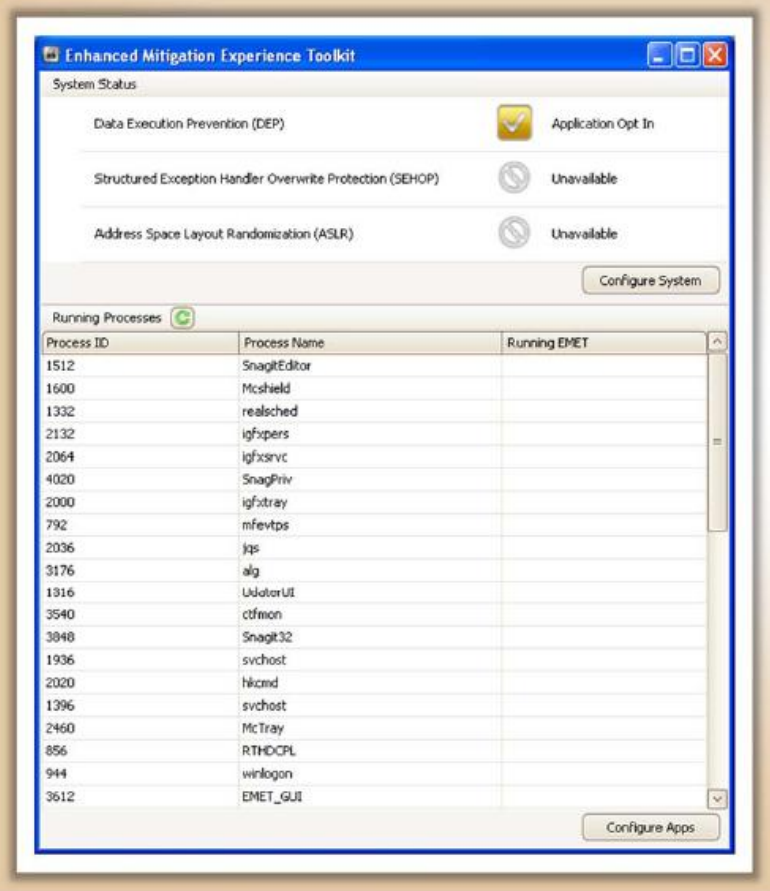


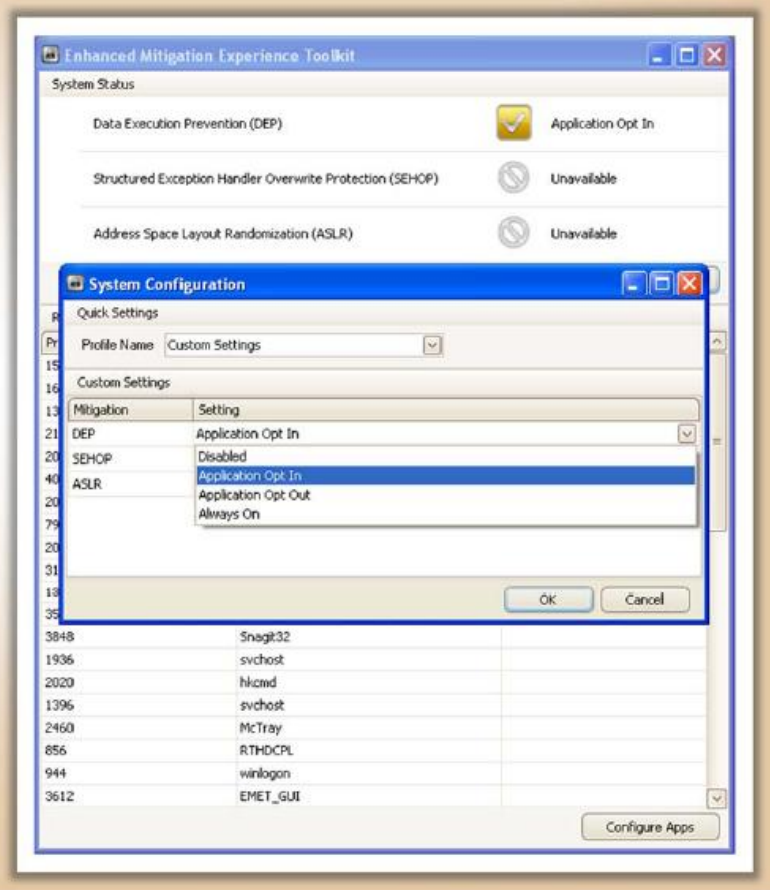
Enhanced Mitigation Experience Toolkit (EMET)

- Enhanced Mitigation Experience Toolkit (EMET) is designed to make it **more difficult** for an attacker to **exploit vulnerabilities** of a software and gain access to the system
- It supports **mitigation** techniques that prevents 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

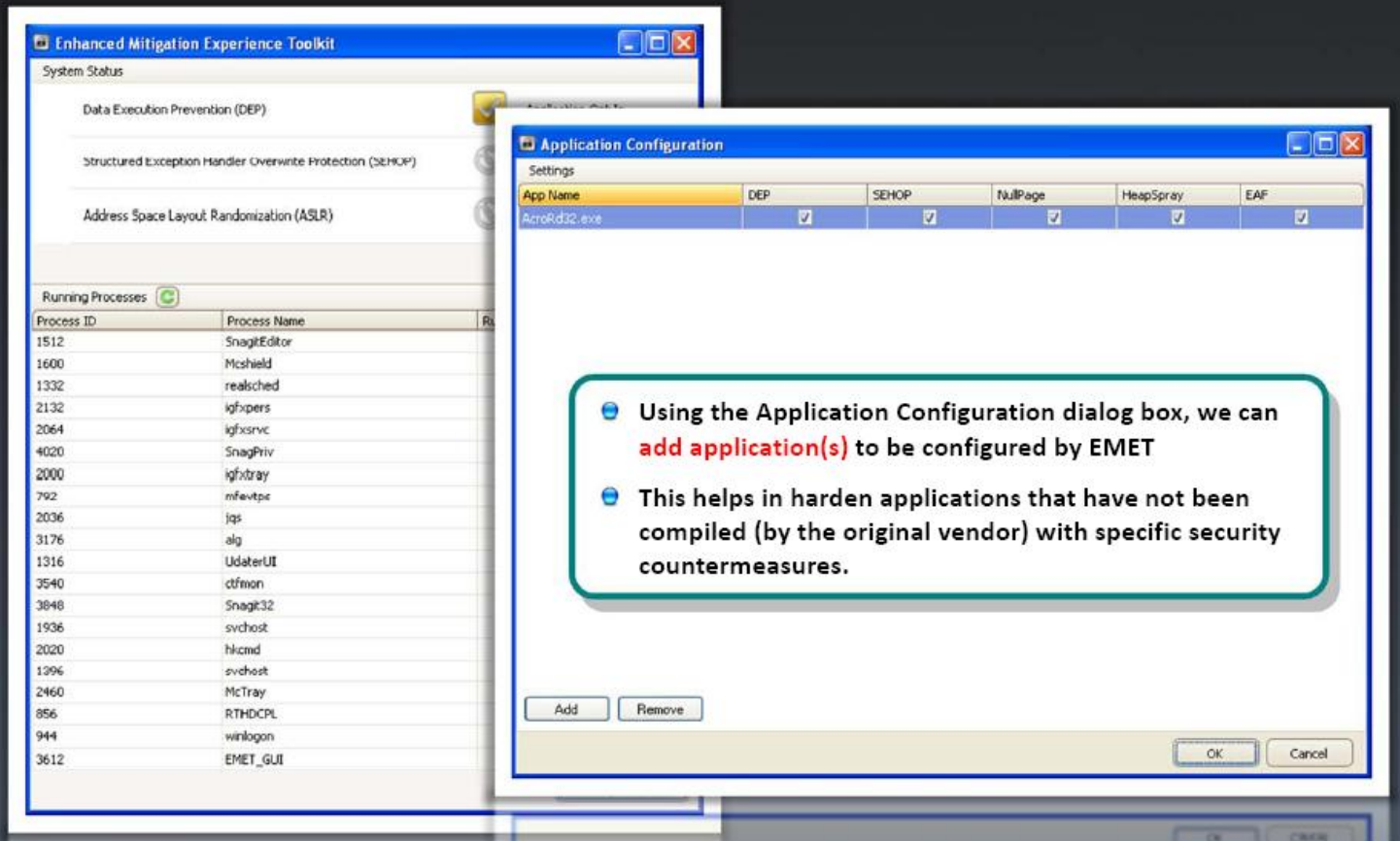
It prevents common techniques used for exploiting stack overflows in Windows by performing SEH chain validation	It marks portions of a process's memory non-executable, making it difficult to exploit memory corruption vulnerabilities	New in EMET 2.0 is mandatory address space layout randomization (ASLR), as well as non-ASLR-aware modules on all new Windows Versions

EMET System Configuration Settings

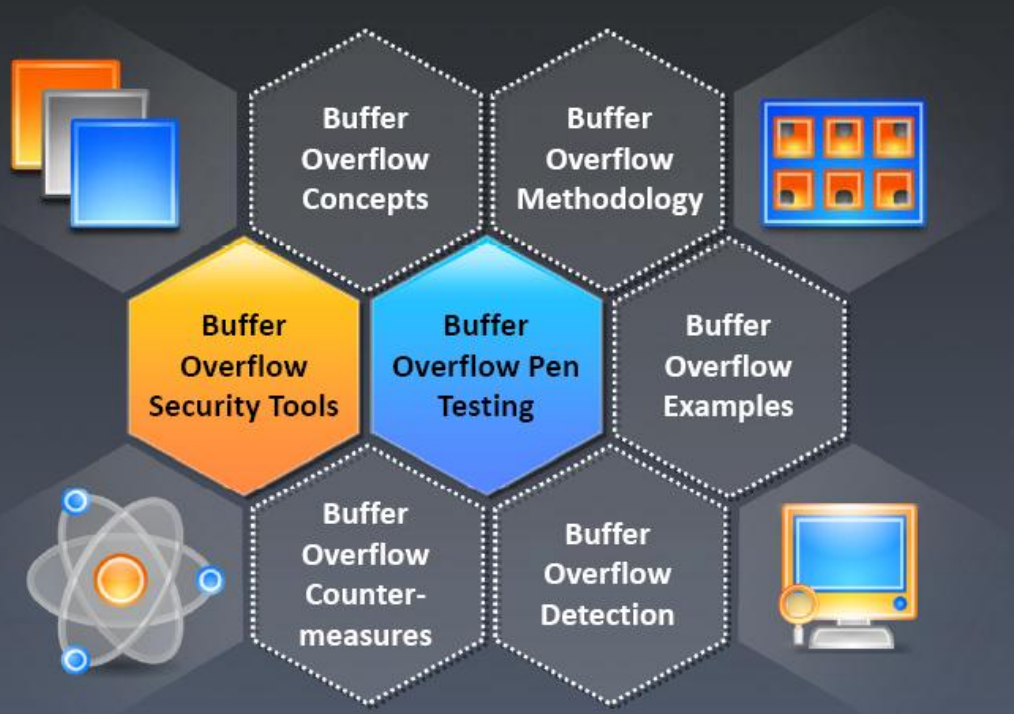




EMET Application Configuration Window



Module Flow



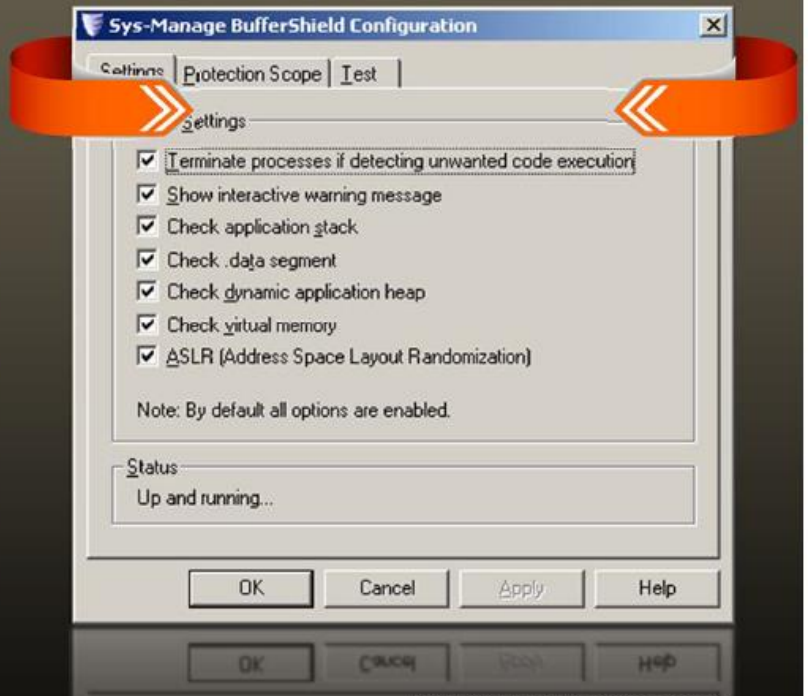
/GS

http://microsoft.com

- Buffer overrun attack utilizes **poor coding practices** that programmers adopt when writing and handling the C and C++ string functions
- /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:

BoF Security Tool: BufferShield

- BufferShield allows you to detect and prevent the **exploitation of buffer overflows**, responsible for the majority of security related problems
- **Features:**
  - **Detects code execution** on the stack, default heap, dynamic heap, virtual memory and data segments
  - **Terminate applications in question** if a buffer overflow was detected



BoF Security Tools

DefencePlus
http://www.softsphere.com

PaX (PAGEEXEC)
http://pax.grsecurity.net

TIED
http://www.security.iitk.ac.in

Clang Static Analyzer
http://clang-analyzer.llvm.org

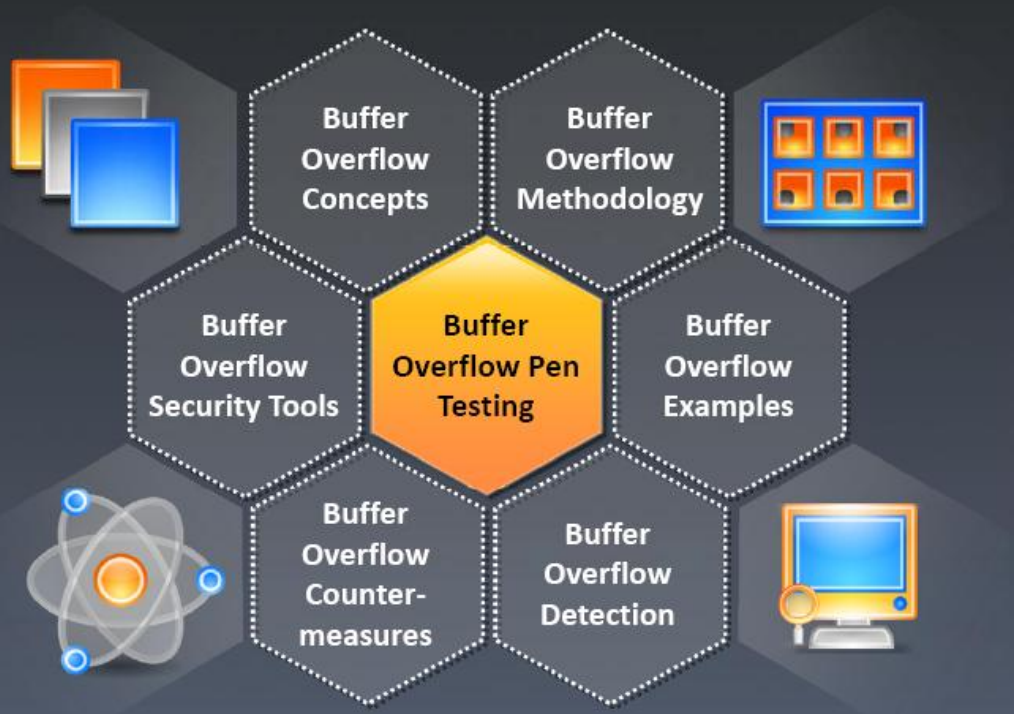
LibsafePlus
http://www.security.iitk.ac.in

FireFuzzer
http://code.google.com

Comodo Memory Firewall
http://www.comodo.com

BOON
http://www.cs.berkeley.edu

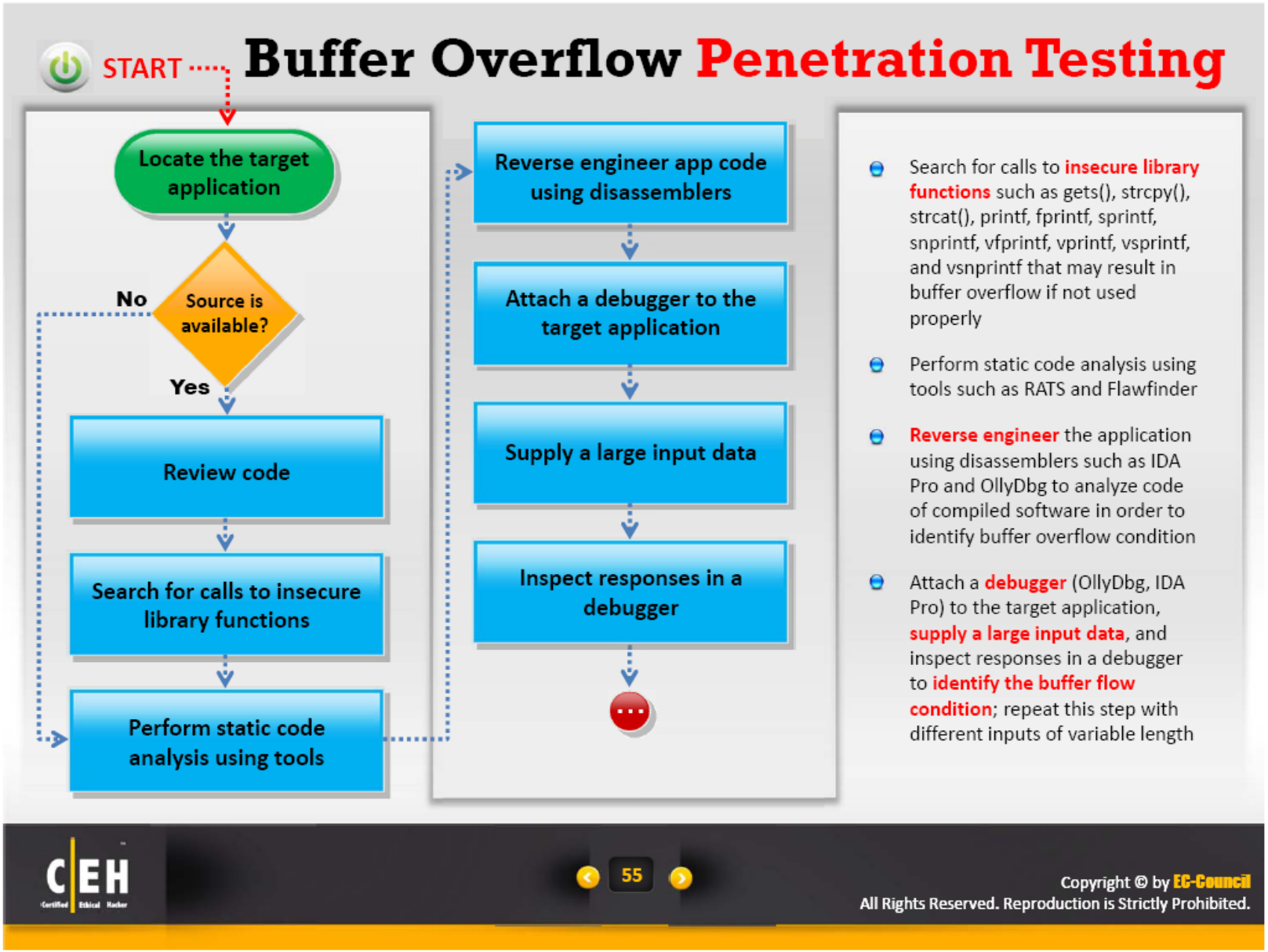
Module Flow



Buffer Overflow Penetration Testing

Buffer overflow penetration testing is based on the assumption that the application will result in system crash or extraordinary behavior when supplied with format type specifiers and input strings that are longer than expected

Buffer Overflow Penetration Testing



- Search for calls to **insecure library functions** such as gets(), strcpy(), strcat(), printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, and vsnprintf that may result in buffer overflow if not used properly
- Perform static code analysis using tools such as RATS and Flawfinder
- **Reverse engineer** the application using disassemblers such as IDA Pro and OllyDbg to analyze code of compiled software in order to identify buffer overflow condition
- Attach a **debugger** (OllyDbg, IDA Pro) to the target application, **supply a large input data**, and inspect responses in a debugger to **identify the buffer flow condition**; repeat this step with different inputs of variable length

Buffer Overflow Penetration Testing

Supply format type specifiers in the input

- Supply **format type specifiers** in the input such as %x or %n
- Use **fuzzing techniques** that provide invalid, unexpected, or random data to the application inputs and observe application behavior

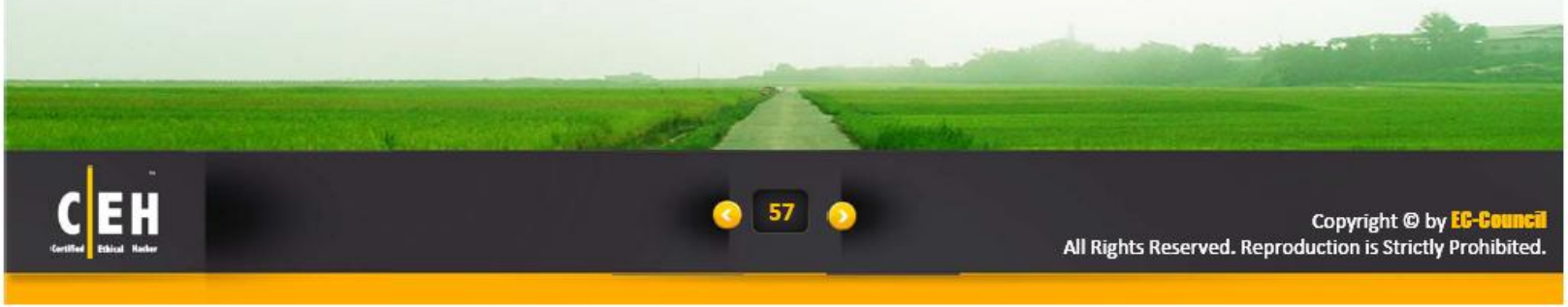
Use fuzzing techniques to overflow the application

- Use fuzzing tools such as **Spike** and **Brute Force Binary Tester (BFB)** for automated fuzzing testing
- Any extraordinary application behavior or **crash** indicates a successful buffer overflow attack

Document all the findings

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
- Buffer overflow vulnerability can be detected by skilled auditing of the code as well as boundary testing
- 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



Quotes

"Design is not just what it looks like and feels like. Design is how it works."

- **Steve Jobs** ,
CEO, Apple Inc.