BASIC VULNERABILITIES DEMONSTRATED IN C AND ASSEMBLY

ANDREW SYVERSON

BUFFER OVERFLOW

BUFFER OVERFLOW - DESCRIPTION

- A buffer overflow occurs when data exceeds the buffer's storage capacity, overwriting adjacent memory.
- A simple C program reading user input without limiting the number of characters to the specified buffer.
- The scanf_s("%s", string) reads a string from the user without limiting the number of characters – if the user inputs more than 49 characters it will overflow the buffer.

```c
#include <stdio.h>

void buffer_overflow() {
    char string[50];

    printf("Enter some text: ");
    scanf_s("%s", string);

    printf("You entered: %s\n", string);
}

int main() {
    buffer_overflow();
    return 0;
}
```

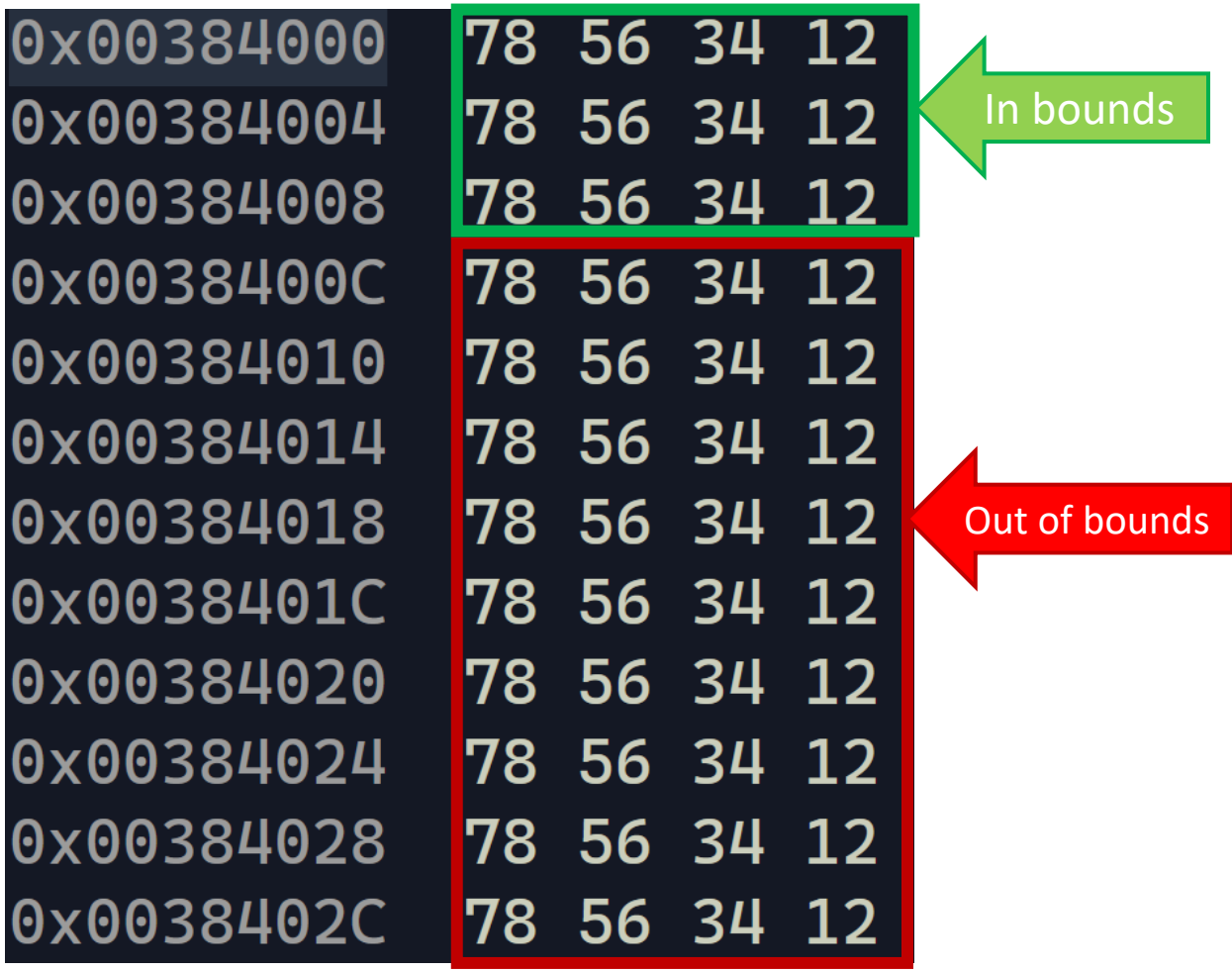
BUFFER OVERFLOW – ASSEMBLY

- Different functionality then the c program
- Starts by initializing a 12-byte buffer filled with 0s
- Then initializes a larger DWORD variable which is 256 bytes in length
  - Why DWORD 64 dup(12345678h)?
    - Dup is a directive that duplicates a chunk of data in this case we are creating 64 DWORDs with that value giving us an array that is 256 bytes long.
- The program then copies the large_data DWORD into the buffer without checking the bounds causing a buffer overflow

```
.386
.model flat, stdcall
.stack 4096

.data
buffer byte 12 dup(0)       ; Define a buffer with 12 bytes filled with 0
large_data DWORD 64 dup(12345678h) ; Define larger data with 64 DWORDs

.code
main PROC
    ; Copy large_data to buffer without bounds checking
    lea esi, large_data
    lea edi, buffer
    mov ecx, 64             ; Set ECX to 64 to copy 64 DWORDs (256 bytes)
    rep movsd               ; Copy DWORDs from large_data to buffer

    ; Exit process using DOS interrupt
    mov ax, 4C00h
    int 21h
main ENDP
END main
```

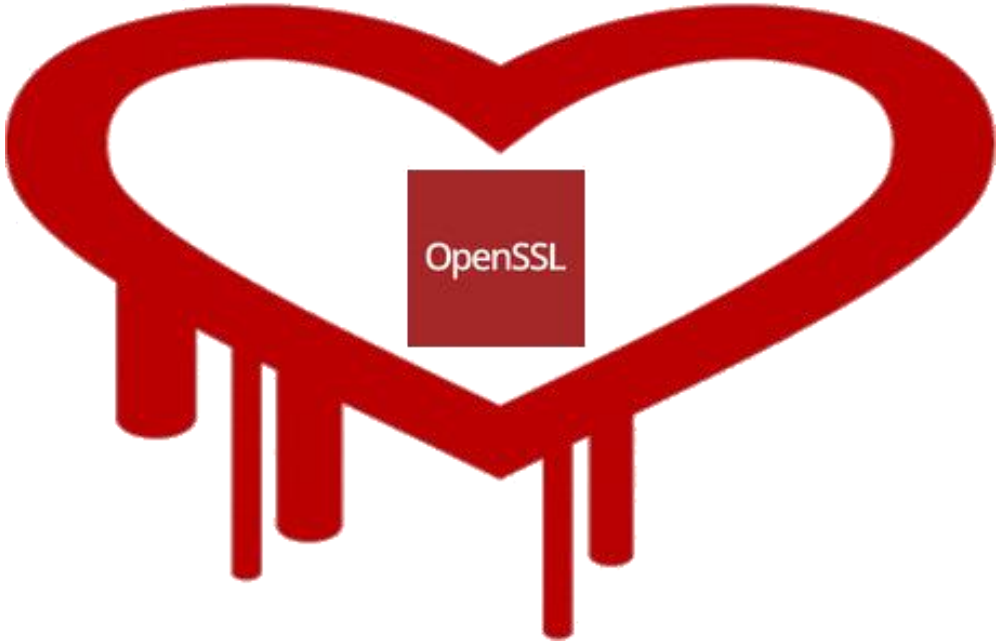
BUFFER OVERFLOW – ASSEMBLY CONTINUED

- Now let's look at the buffer overflow in memory
- Get the starting memory address of “buffer” by searching “&buffer” in memory
  - Starting address is 0x00384000
- Why are we seeing the pattern 78, 56, 34, 12?
  - The array we initialized with the repeating value “12345678h”
  - If you break that hex value down byte by byte you see the pattern 12, 34, 56, 78



BUFFER OVERFLOW – IMPACT

- Sensitive Data Exposure: Buffer overflow vulnerabilities can lead to leakage of sensitive information, including passwords, credit card numbers, and other personal data.
- System Compromise: Attackers exploiting buffer overflow vulnerabilities can gain unauthorized access, potentially taking control of the system.
- Example: Heartbleed, a buffer overflow bug in OpenSSL, allowed attackers to read memory contents from servers, exposing critical data and affecting millions of systems worldwide.



https://heartbleed.com/

USE-AFTER-FREE

USE-AFTER-FREE - DESCRIPTION

- A use-after-free vulnerability occurs when a program continues to use a pointer after it has been freed.
- A simple c program that uses a pointer after the memory has been freed
- Initializes the pointer and assigns it the value 42 and allocates memory for it. Then frees the memory and later calls the pointer in the print statement after being freed.

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

void use_after_free() {
    int* ptr = (int*)malloc(sizeof(int));
    *ptr = 42; // Allocate and assign a value

    free(ptr); // Free the memory

    // Vulnerable: using the pointer after it has been freed
    printf("Value after free: %d\n", *ptr);
}

int main() {
    use_after_free();
    return 0;
}
```

USE-AFTER-FREE – ASSEMBLY

- This assembly program is really doing three things to simulate a use-after-free vulnerability

1. First “lea esi, buffer” loads the address of buffer into the esi register. Then the instruction “mov eax, [esi] moves the value at the address ESI into the EAX register. Now the EAX register holds the value 5.
2. Second it simulates freeing the memory by moving 0 into the address ESI (memory address of buffer) to simulate freeing the memory.
3. Finally, it attempts to access the value at that memory address which we just freed (set to 0) and sets the EBX register to that value.

```
.386
.model flat, stdcall
.stack 4096

.data
buffer dd 5                    ; Allocate 4 bytes (1 DWORD) with value 5
msg db 'Freed memory accessed', 0

.code
main PROC
    ; Simulate freeing memory by using a different part of the buffer
    lea esi, buffer
    mov eax, [esi]             ; Move the value 5 into EAX

    ; Free memory
    mov dword ptr [esi], 0     ; Clear the value to simulate freeing

    ; Use-after-free (vulnerable)
    mov ebx, [esi]             ; Access the freed memory (should be 0)

    ; If a debugger is used, ebx will hold the value from freed memory

    ; Exit process using DOS interrupt
    mov ax, 4C00h
    int 21h
main ENDP
END main
```

USE-AFTER-FREE – ASSEMBLY CONTINUED

- Now let's look at those operations by looking at the registers in debug mode – set a break point at lines 14, 17, and 20 to see all three steps

1. The first operation on line 14 we load 5 into the EAX register by pointing to its location in memory (0x00FA4000).
2. The second break point at 17 now executes – where we free the memory by setting that memory address to 0 notice how 5 is still stored in EAX register but the memory location is now 0.
3. The third and final breakpoint on line 20 points to that location in memory. It then tells the computer to fetch the data located there and move it into the EBX register – So the EBX register is 0 since we just set that location in memory to 0 (freed it)
    a. In real life this would cause the computer to point to a random value or crash the program.

```
EAX = 00000005 EBX = 00B9E000 ECX = 00FA1005 EDX = 00FA1005
  ESI = 00FA4000 EDI = 00FA1005 EIP = 00FA1018 ESP = 00D5F880
  EBP = 00D5F88C EFL = 00000246
```

```
12      lea esi, buffer
13      mov eax, [esi]            ; Move the value 5 into EAX
14
15      ; Free memory
16      mov dword ptr [esi], 0    ; Clear the value to simulate freeing
17
18      ; Use-after-free (vulnerable)
19      mov ebx, [esi]            ; Access the freed memory (should be 0)
20
21      ; If a debugger is used, ebx will hold the value from freed memory
```

```
EAX = 00000005 EBX = 00000000 ECX = 00FA1005 EDX = 00FA1005
  ESI = 00FA4000 EDI = 00FA1005 EIP = 00FA1020 ESP = 00D5F880
  EBP = 00D5F88C EFL = 00000246
```

USE-AFTER-FREE – IMPACT

- Unpredictable Program Behavior: Use-after-free vulnerabilities can cause programs to behave unpredictably, including crashes or unexpected results, as they may access memory that has been reallocated for other uses.
- Arbitrary Code Execution: Attackers exploiting use-after-free vulnerabilities can execute arbitrary code by manipulating the freed memory, potentially leading to full system compromise.
- Example: CVE-2024-4671 - Use after free vulnerability in Visuals in Google Chrome (prior to 124.0.6367.201) allowed a remote attacker who had compromised the renderer process to potentially perform a sandbox escape via a crafted HTML page. (Chromium security severity: High)

https://hackersonlineclub.com/analysis-cve-2024-4671-zero-day-in-google-chrome/

https://nvd.nist.gov/vuln/detail/CVE-2024-4671


THANK YOU