IT Security Principles:

Linux Exploitation

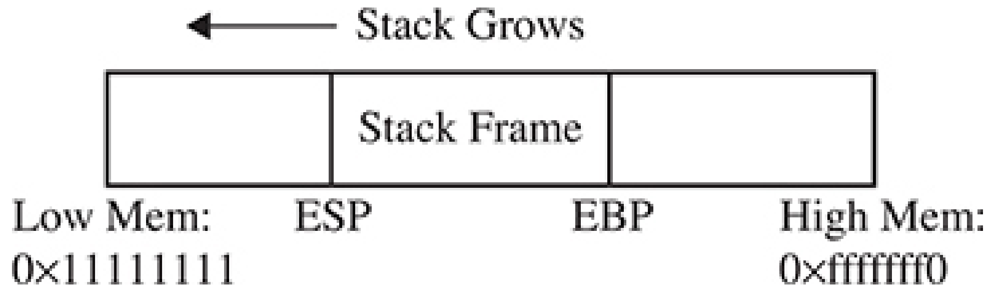
IT 444 – Network Security

Stack Operation

- The concept of a stack in computer science can best be explained by comparing it to a stack of lunch trays in a school cafeteria
- The process of putting items on the stack is called a push
- Taking an item from the stack is called a pop (pop command in assembly language code)
- Every program that runs has its own stack in memory
- The stack grows backward from the highest memory address to the lowest. (bottom -> top)

Stack Operation

- Two important registers deal with the stack:
  - Extended Base Pointer (EBP)
  - Extended Stack Pointer (ESP)
- EBP register is the base of the current stack frame of a process (higher address).
- ESP register always points to the top of the stack (i.e., lower address).



Stack Operation

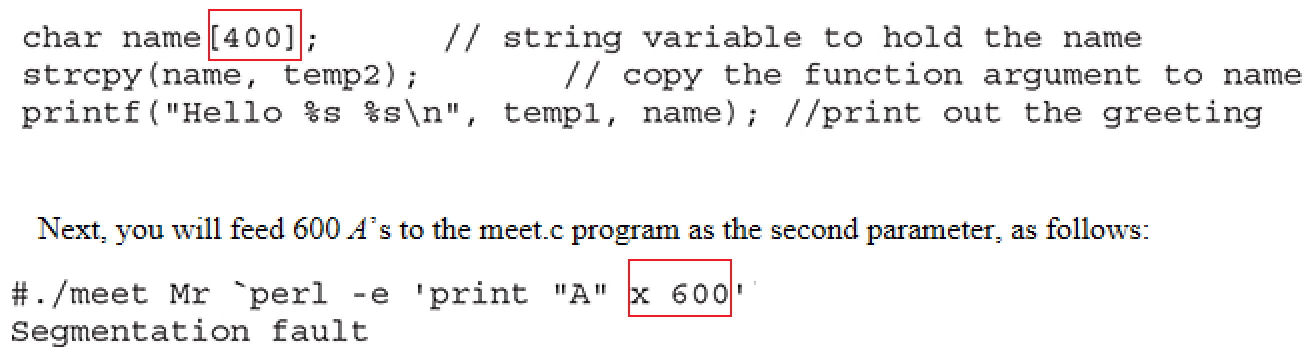
- When a function is called in assembly code, three things take place:
    - The calling program sets up the function call by first placing the function parameters on the stack in reverse order.
    - The Extended Instruction Pointer (EIP) is saved on the stack so the program can continue where it left off when the function returns (the *return address*).
    - The **call** command is executed, and the address of the function is placed in the EIP to execute

Stack Operation

- The called function's responsibilities are…
  - First, to save the calling program's EBP register on the stack,
  - Next, to save the current ESP register to the EBP register (setting the current stack frame)
  - Then, to decrement the ESP register to make room for the function's local variables.
  - Finally, the function gets an opportunity to execute its statements.
  - The last thing a called function does before returning to the calling program is to clean up the stack by incrementing ESP to EBP -- clearing the stack as part of the **leave** statement

Buffer Overflows

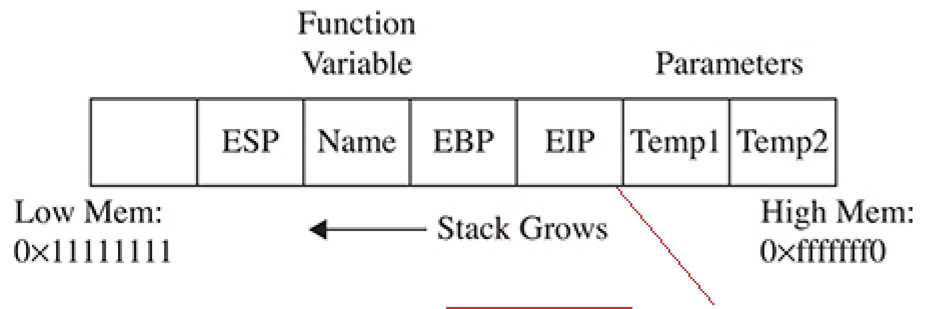
- Buffers are used to store data in memory
- Buffers *cannot* prevent you from putting too much data into the reserved space



```
char name[400];       // string variable to hold the name
strcpy(name, temp2);        // copy the function argument to name
printf("Hello %s %s\n", temp1, name); //print out the greeting
```

Next, you will feed 600 *A*'s to the meet.c program as the second parameter, as follows:

```
#./meet Mr `perl -e 'print "A" x 600'`
Segmentation fault
```



If you write past the EIP, you will overwrite the function arguments, starting with **temp1**.

Ramifications of Buffer Overflows

- In a buffer overflow, three things can happen:
  1. Denial of service: Easy to get a segmentation fault when dealing with process memory
  2. EIP can be controlled to execute malicious code at the user level of access. This happens when the vulnerable program is running at the user level of privilege
  3. EIP can be controlled to execute malicious code at the system or root level

Local Buffer Overflow Exploits

- Local exploits are easier because you have access to the system memory space and debug exploits easier.
- The basic goal of buffer overflow exploits is to overflow a vulnerable buffer and change the EIP for malicious purposes.
  - the EIP points to the next instruction to be executed. An attacker could use this to point to malicious code
  - A copy of the EIP is saved on the stack to continue with the command after the call when the function completes
  - Influencing the saved EIP value, when the function returns, the corrupted value of the EIP will be popped off the stack into the register (EIP) and then executed.

Local Buffer Overflow

- In assembly, **NOP** means to do nothing but move to the next command
- When placed at the front of an exploit buffer, this padding is called a "NOP sled".
- If the EIP is pointed to a NOP sled, the processor will ride the sled right into the next component
- On **x86** systems, the **0x90** opcode represents **NOP**.
- The most important element of the exploit is the return address, which must be aligned perfectly and repeated until it overflows the saved EIP value on the stack

Exploit development

- The exploit development process generally follows these steps:
  1. Control the EIP.
  2. Determine the offset(s).
  3. Determine the attack vector.
  4. Build the exploit.
  5. Test the exploit.
  6. Debug the exploit, if needed.

Format String Exploits

- Format string exploits became public in late 2000, they are still common in applications today
- Many organizations still don’t utilize code analysis or binary analysis tools on software before releasing it
- Format strings are used by various print functions
  - When someone calls one of these functions, the format string dictates how and where the data is compiled into the final string
  - If the creator of the application allows data specified by the end user to be used directly in one of these format strings, the user can change the behavior of the application.

Format String Exploits

- If the programmer is sloppy and does not supply the correct number of arguments, or if the user is allowed to present their own format string, the function will happily move down the stack (higher in memory), grabbing the next value to satisfy the format string requirements) – including disclosing as memory locations, data variables, and stack memory

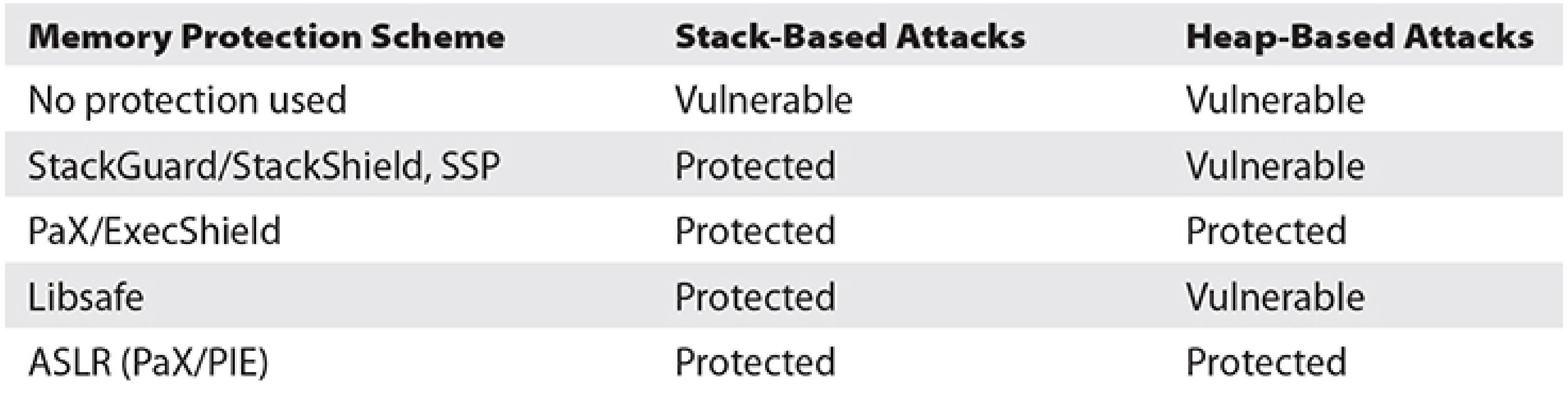
Memory Protection Schemes

- Since buffer overflows and heap overflows have come to be, many programmers have developed memory protection schemes to prevent these attacks
- Libsafe is a dynamic library that allows for the safer implementation of the following dangerous functions:
  - strcpy(), strcat(), sprintf()   and vsprintf()
  - getwd(), gets(), realpath(), fscanf(), scanf(), and sscanf()
- Libsafe overwrites these dangerous libc functions by replacing the bounds and input-scrubbing implementations, thereby eliminating most stack-based attacks

Memory Protection Scheme

- StackShield, StackGuard, and Stack Smashing Protection (SSP)
  - A replacement to the **gcc** compiler that catches unsafe operations at compile time
  - When a function is called, StackShield copies the saved return address to a safe location and restores the return address upon returning from the function
- StackGuard: If a buffer overflow attempts to overwrite the saved EIP, the canary will be damaged and a violation will be detected
- Stack Smashing Protection (SSP): rearranging the stack variables to make them more difficult to exploit

Memory Protection Scheme



Memory Protection Scheme	Stack-Based Attacks	Heap-Based Attacks
No protection used	Vulnerable	Vulnerable
StackGuard/StackShield, SSP	Protected	Vulnerable
PaX/ExecShield	Protected	Protected
Libsafe	Protected	Vulnerable
ASLR (PaX/PIE)	Protected	Protected