Introduction to Information Security

Dr. Ashok Kumar Das

Center for Security, Theory and Algorithmic Research
International Institute of Information Technology, Hyderabad

E-mail: *ashok.das@iiit.ac.in*
URL: http://www.iiit.ac.in/people/faculty/ashokkdas
Personal Home Page: http://sites.google.com/view/iitkgpakdas/

Buffer Overflow (BoF) Attacks

Stack-related Preliminaries

- Each process is allocated separate space for its code and data.
- The instructions in a program are assigned to the code or text segment.
- Separate segments are assigned for static initialized and global uninitialized data.
- Another segment is dedicated to the stack and heap.
- To save space, the stack and heap grow in opposite direction.
- Program stack is used to store the local variables, while the heap is used to store the dynamically created variables.

Stack-related Preliminaries

Table: Organization of process memory

CODE	0000 0000
GLOBAL UNINITIALIZED DATA	
STATIC INITIALIZED DATA	
HEAP ↓ ⋮ ↑ STACK	FFFF FFFF

Stack-related Preliminaries

- A stack is a Last in First out (LIFO) data structure.
- When a program calls a function (subroutine), a stack frame for the called function is created.
- The stack frame is used to save the state of a calling program.

```
 Function A                Function B
(Calling program)      (Called program)
------------------------------------------------

int A ( ) {         void B (int k){
   B(29);             char buffer[100];
   return 0;          int j = 17 + k;
}                     printf("Enter your name:");
                      gets(buffer);
                      printf("Hello %s", buffer);
                      return;
                    }
```

Stack-related Preliminaries

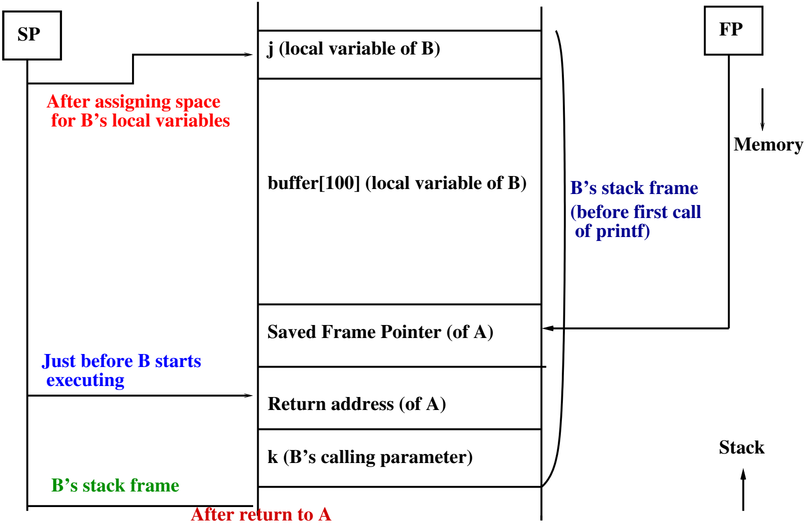

SP
j (local variable of B)
After assigning space for B's local variables
buffer[100] (local variable of B)
Saved Frame Pointer (of A)
Just before B starts executing
Return address (of A)
k (B's calling parameter)
B's stack frame
After return to A
FP
Memory
B's stack frame (before first call of printf)
Stack

Figure: Stack frame for calling and called programs

Stack-related Preliminaries

- A processor has many registers. Some of these are general-purpose registers used to store variables in a program.
- On Intel Pentium machines, for example, these are 32-bit registers
  - EAX: Accumulator register
  - EBX: Base register
  - ECX: Count register
  - EDX: Data register
  - ESI: Source index
  - EDI: Destination index
  - EBP: Base pointer
  - ESP: Stack pointer
- Three special-purpose registers
  - FP: Frame pointer
  - IP: Instruction pointer (Program counter)
  - SP: Stack pointer

Stack-related Preliminaries

- The IP points to the next instruction to be fetched.
  When B() completes, the control is returned to the calling program A(). The return address was PUSHed on the stack by the call instruction in A. The last instruction in B (usually ret or return) will POP A's return address of the stack and into the IP.
- The FP points to a fixed location in the stack frame of the currently executing subroutine.
  All local variables in B as well as arguments passed to it (by A) are referenced as displacements from the FP.
- The SP points to the top of the stack, i.e., the last item PUSHed on the stack.
  The SP also keeps changing value as local variables are pushed on the stack during the execution of the subroutine.

Why does Buffer Overflow (BoF) occur in first place?

- Programs written in languages such as C/C++ are especially susceptible to BoF attacks.
- The C language was written with efficiency and flexibility in mind.
- C/C++ are *loosely-type languages* which allow pointer arithmetic and do not perform array bound checking.
- Consider the following C function

  ```
  gets(char * buffer);
  ```

  reads a string from an I/O stream until it finds a newline character (NULL character). It does not check whether the input string is within the size limits of the destination buffer being populated.

Why does Buffer Overflow (BoF) occur in first place?

- There are many such C functions:

```
char *strcpy (char *dest, const char *src);
char *strcat (char *dest, const chat *src);
int sprintf(char *str, const char *format, ...);
int vsprintf (char *str, const char *format,
              va_list ap);
int scanf (const char *format, ...);
int sscanf (const char *str, const char *format,
            ...);
int fscanf (FILE *stream, const char *format,
            ...);
int vscanf (const char *format, va_list ap);
int vsscanf (const char *str, const char *format,
             va_list ap);
int vfscanf(FILE *stream, const char *format,
            va_list ap);
```

Key points regarding the BoF vulnerability

- The stack and memory (heap) grow in opposite directions.
  A local variable such as buffer is written into starting low to high addresses in memory. So, if buffer overflows, it will corrupt the contiguous areas of the stack which includes the return address.
- Suppose the input to buffer is derived from an external source.
  An attacker can craft the input string copied into buffer so that the return address of the calling program A() is overwritten to point to malicious code.
  On completion of the called program B(), control will return not to A() but to a location in memory determined by the attacker.
- The malicious code could itself be included in buffer. Alternatively, the return address could be overwritten to point to a library function, which creates a shell.
  The attacker's string could also contain the necessary arguments required by the library function.

Exploiting Stack Overflows

Exploit Number 1: Use of shellcode

- Consider a buffer that accepts input from an external source -
  - a keyboard, or
  - the payload of a network packet.
- One exploit is through malicious code injection - placing malicious code in an array variable of the vulnerable program.
- The malicious code is commonly referred to as “shellcode” since the most obvious exploit involves spawning a shell.
- To obtain the services of an OS, a program makes a “system call”. There are system calls to open a file, create a process, etc.
- Each system call has an associated number. For example, on Linux platforms, system call $\#5$ creates/opens a file, and system call $\#8$ creates a new process.

Exploiting Stack Overflows

Exploit Number 1: Use of shellcode

- To make a system call on Linux running on an Intel ×80 processor, the system call number is placed in the EAX register and a software interrupt is generated using the sequence of instructions:

```
mov EAX, 11 // System call to execve ()
int 0x80    // Software Interrupt No. 80
```

- The software interrupt generates a signal to the kernel, which invokes a call handler.
- In addition, the system call parameters need to be passed.
- On Linux systems, these are passed through registers. System call #11 used in the above shellcode is (execute program)

```
int execve (const char *filename,
            const char *argv[],
            const char *env[]);
```

It replaces the image of the calling process with a new process image.

Exploiting Stack Overflows

Exploit Number 1: Use of shellcode

- The shellcode is injected into a buffer of the vulnerable program through input received directly or indirectly from the attacker.
- Problem: How does the attacker ensure that the shellcode gets executed?
- The attacker must be able to overwrite the return address on the stack with the address of the shellcode.
- But how does the attacker know the address of the shellcode?

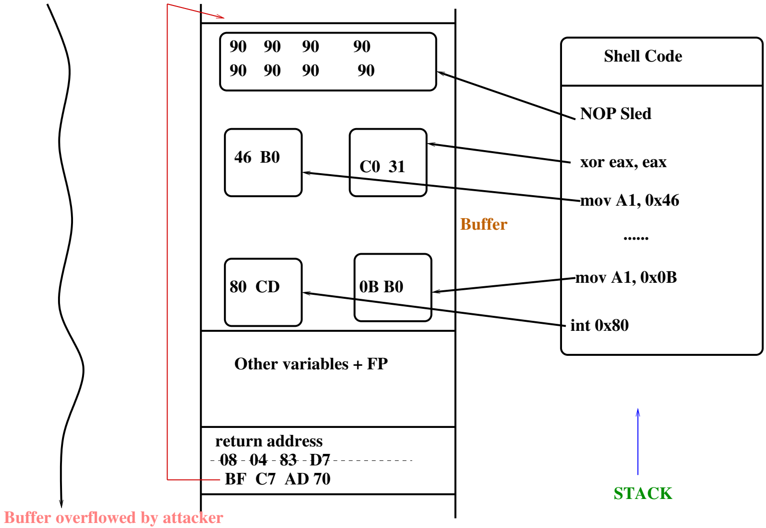
Exploiting Stack Overflows

Exploit Number 1: Use of shellcode

- It is important for the attacker to have precise knowledge of the program stack on the vulnerable platform.
- This is not always straightforward but persistence on the part of the attacker can play rich dividends.
- By experimentation, the attacker may be able to deduce the address of the top of the vulnerable program's stack frame, the address of the buffer containing the shellcode, and the location on the stack where the calling program's return address is saved.
- This enables the attacker to not only inject shellcode into the buffer but also overwrites the return address with the start address of the shellcode.

Exploiting Stack Overflows

Exploit Number 1: Use of shellcode



Exploiting Stack Overflows

Exploit Number 2: Return-into-LibC

- This exploit uses existing code in the C library to spawn a shell.
- LibC is a shared library of functions such as *printf* and other functions for file access, math, etc.
- Every C language program in execution is linked to LibC.
- Moreover, the start address of each library function on a particular OS is usually fixed and can easily be determined.

Exploiting Stack Overflows

Exploit Number 2: Return-into-LibC

- This exploit makes a call to the library function, *system()*, which internally invokes the system call *execve()*.
- The exploit contains in transferring control to *system()* with parameter “\bin\sh”:
  system(“\bin\sh”)

Exploiting Stack Overflows

Exploit Number 2: Return-into-LibC

- Let *B* be the program with the BoF vulnerability (*A* is the calling program and *B* is the called program).
- The attacker overflows a buffer in *B*'s stack frame so that the return address of *A* is overwritten by the address of the library function, system().
- As always, when *B* exists, it pops the return address on to the IP or the PC (program counter).
- However, in this case, the attacker overwrites the "return address" with the address of system()!

Exploiting Stack Overflows

Exploit Number 2: Return-into-LibC

- The system() function thinks that it has been invoked through a regular sub-routine call.
- So, it assumes that the SP (Stack Pointer) is pointing to the caller's return address.
- As part of the BOF exploit, the attacker stores the address of the function, exit(), in this location.

Exploiting Stack Overflows

Exploit Number 2: Return-into-LibC

In summary, the Return-into-LibC exploit overflows the buffer so that

- the saved return address is overwritten by the address of system().
- the address of exit() is placed on the stack below the address of system().
- the string “\bin\sh” is loaded somewhere in memory.
- the address of the above string is placed on the stack just below the address of exit().

Exploiting Stack Overflows

Note

- Note that, unlike the previous exploit “Use of shellcode”, this exploit “Return-into-LibC” does not inject any malicious code.
- Instead, it invokes a C library function to spawn a shell.

Exploiting Stack Overflows

Defenses of BOF vulnerability

- BOF have been contemplated at various levels- at the level of the
  - program/programming language
  - compiler
  - operating system
  - hardware

Exploiting Stack Overflows

Defenses of BOF vulnerability

- To minimize the chance of a successful BOF exploit, the programmer could develop his/her application in a type-safe language such as Java or C#.
- The C and C++ languages are widely used for reasons of performance and flexibility, If those or other reasons, a programmer must use C/C++, there are a number of precautions he/she could take.
- One recommendation is to avoid the use of dangerous functions such as *gets()*, *strcpy*, *sprintf*.
- Instead, their “safe” counterparts- fgets(), strncpy() and snprintf() should be used.
- For example,

  ```
  char *strncpy (char *destination, const char *source, size_t count);
  ```

Exploiting Stack Overflows

Defenses of BOF vulnerability

- Having C/C++ code audited can help to reduce the number of BOF vulnerabilities.
- There are a number of automated tools that perform static analysis, identify dangerous functions, and warn programmers of suspicious code sequences.

Exploiting Stack Overflows

Defenses of BOF vulnerability

- Many operating systems like Windows XP SP2, several Linux variants, etc., make the stack non-executable.
- This frustrates code-injection attacks (where attack code is placed on the stack).
- However, this does not preclude Return-into-LibC attacks.
- Also, there are some legacy applications that place executable code on the stack.
- So, making the stack non-executable is NOT always a practical solution.

Exploiting Stack Overflows

Defenses of BOF vulnerability

- One popular approach at the compiler level is the use of a 32-bit random number (called a canary).
- Compiler-generated code is included in the function prologue to place the canary between the FP and the return address.

Local variables
Saved FP
Canary
Return address
Function parameters

- Just before the function returns, the canary is checked to determine whether its value has changed.

Exploiting Stack Overflows

Defenses of BOF vulnerability

- Another proposed solution to BOF is to randomize the layout of memory.
- Here, the entry point to library functions, base address of stack, etc. are randomly assigned within limits.
- In such case, the exploits may not be successful.

Exploiting Stack Overflows

Defenses of BOF vulnerability

- Other solutions include the use of safe C compilers or safe libraries that check memory addresses at run-time.
- However, these solutions typically incur unacceptable performance overheads.
- Finally, hardware solutions such as use of a register (rather than the stack) to store return addresses have also been proposed.

Format String Attacks

Format String Attacks

- C functions such as **printf( )** take a format string as the first argument as in printf(“var1 in decimal is % d and var2 in hex is % x \n”, var1, var2).
- %*d* and %*x* are referred to as format specifiers.
- Each occurrence of a %*d* or %*x*, printf( ) expects to see a variable name in its list of arguments.
- Likewise, for each occurrence of a %*s* or %*n*, printf( ) expects to see pointer arguments.
- For example, printf(“This course name is %*s*”, name).
  Here name is a string variable, i.e., pointer to an array of characters.
- As it turns out, %*s* and %*n* are the format specifiers that can be used by an attacker to read and write arbitrary locations in memory.

Format String Attacks

Example

```
printf("# of characters printed is %n", &count);
printf("%d", count);
```

- The first printf( ) statement causes the number of bytes printed by that statement to be output to the variable, count.
- The second printf( ) statement then displays the number of bytes printed.
- In the above program, output is 27.

Format String Attacks

Reading from an arbitrary memory location

```
int main (int argc, const char * argv[]) {
    char buf[160];
    strcpy(buf, argv[1]);
    printf(“The content of memory location 12345678 is: ”);
    printf(buf);
    exit(0);
}
```

- If we compile and execute the above program with the following command-line input:
  ./a.out “Hello %d”
  we get the output as
  The content of memory location 12345678 is: Hello 134514137

SQL Injection Attacks

SQL Injection

The Vulnerability

- Multi-tier web applications typically have three tiers- the web, application, and database tiers.
  - The Web tier interacts directly with the application layer which, in turn, interfaces with the database tier.
    For applications with limited business logic, the application tier may be fused with the web tier.
    In such applications, a component in the web tier may directly communicate with the database tier.
  - The database tier contains the database and the Database Management System (DBMS).
    The database is itself compromised of a number of tables (or relations).
    Each relation has a number of attributes (columns) and tuples (rows).
    A relation is an instance of a schema.

SQL Injection

The Vulnerability

Table: An instance of the Students schema (students14)

S_ID	Name	Encrypted Password	Department	CGPA
201705514	Jairam P.	3q!yrsj	CSE	9.89
201706617	Rahul S.	Bu73@jT	ECE	8.60
201701179	Priya D.	L189!24	CSE	9.71
⋮	⋮	⋮	⋮	⋮

SQL Injection

The Vulnerability

- Consider a University website that permits its students to view their registration status and grade information provided they login with their correct student ID and password.
- The University web server presents a webpage containing a form to the user (HTML supports forms using a special ⟨ form ⟩ tag).
- The inputs entered by the user are passed to the server as form parameters when the user clicks on the form's SUBMIT button.
- Form parameters may be passed in the body of an HTTP POST request. Alternatively, they may be passed as a query string in an extended URL to the server as http://www.iiit.ac.in?S_ID = 201706617 & passwd = Bu73@jT

SQL Injection

The Vulnerability

- The server application retrieves the form parameters and uses them to build an SQL query such as

```
select   S_ID, CGPA
from     students14
where    S_ID = 201706617 and password = ‘Bu73@jT’
```

- Some applications build SQL queries using string concatenation and then submit the query to the DBMS.

SQL Injection

The Vulnerability

```
String studentID =    . . .; \\ student’d ID obtained from HTTP
                      request packet
String pw =           . . .; \\ student’s password obtained from
                      HTTP request packet
Connection con =      . . .; \\ Connection to the DB obtained
Statement stmt =      con.CreateStatement( );
String query =        “select CGPA
                      from students17
                      where S_ID = ” +studentID + “and pass-
                      word = ‘” + pw + “’”;
ResultSet rs =        stmt.executeQuery(query);
```

SQL Injection Attacks

Attack 1

IIIT Hyderabad

To obtain your most recent CGPA enter your studentID and password below

Student ID: 123

Password: abc’ or ‘x’ = ‘x

The query built from the inputs supplied by the attacker is

```
select   S_ID, CGPA
from     students17
where    S_ID = 123
         and password = ‘abc’ or ‘x’ = ‘x’
```

SQL Injection Attacks

Attack 2

IIIT Hyderabad

To obtain your most recent CGPA enter your studentID and password below

Student ID: 123 or 1 = 1 - -

Password: abc

The query built from the inputs supplied by the attacker is

```
select   S_ID, CGPA
from     students17
where    S_ID = 123 or 1 = 1 - -
         and password = ‘abc’
```

SQL Injection Attacks

Attack 3

IIIT Hyderabad

To obtain your most recent CGPA enter your studentID and password below

Student ID: 123; DROP table students17; - -

Password: abc

The query built from the inputs supplied by the attacker is

```
select   S_ID, CGPA
from     students14
where    S_ID = 123; DROP table students17; - -
         and password = 'abc'
```

SQL Injection

SQL Injection Remedies

- One mitigation strategy is to parse user input and reject input with symbols such as ‘ (single quote), “ double quote), ; (semi-colon), etc.
  However, the input string containing 1 = 1 that appeared in one of the above queries may not get filtered out.
  On the other hand, some names (such as O’Neal) have quotes in them and such user input may be wrongly rejected.
- Another approach is to specify what is acceptable rather than what is not.
  This can be done through the use of regular expressions.

Reference: http://www.unixwiz.net/techtips/sql-injection.html

Thank You !!!