OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Secure Programming

Input Validation

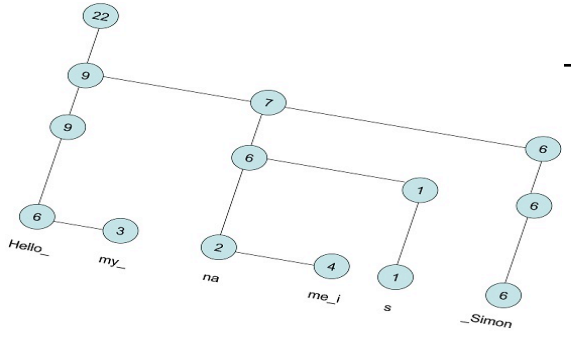
Prof. Dr. Christoph Skornia
christoph.skornia@hs-regensburg.de

Input Validation – Strings

facts about strings:

- not built in type in C and C++
  - C: array of chars
  - C++: ISO/IEC98 is promoting `std::basic_string`, `std::string` and `std::wstring`
  - further structures like rope or cord for storing very long strings
- different definition for string-classes on different OS
  - `std::string` is UTF-8 by default
  - `std::wstring` is dependent on system and compiler
    - `std::wstring` is UTF-16 on Windows
    - `std::wstring` is UTF-32 on (typical) Unix-type systems (like Linux and OSX)
    - use `sizeof(wchar_t)` or list predefined macros (cpp –dM Ctrl+D) to clarify if unsure

Input Validation – Strings



facts about strings:

- null-termination
  - C-style strings are NULL-TERMINATED arrays
  - internal representation of `std::basic_string`, `std::string` and `std::wstring` does not need to be null terminated as well as Win32 `LSA_UNICODE_STRING`
- usually different types of strings are used (especially in legacy code)
  - different libraries require different types of strings
  - different type have different memory requirements
  - type conversions are necessary and may lead to vulnerabilities

Input Validation – Strings

common errors:

- Unbounded stream from stdin
  bad example:

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

better: limit size (truncate)

```
int main(void) {
   char Password[9];
   puts("Enter 8 character password:");
   fgets(Password,9,stdin);
  ...
 }
```

Input Validation – Strings

common errors:

- Unbounded string copy and concatenation example:

```
int main(int argc, char *argv[]) {
   char name[2048];
   strcpy(name, argv[1]);
   strcat(name, " = ");
   strcat(name, argv[2]);
 ...
 }
```

better: dynamic allocation

```
int main(int argc, char *argv[]) {
   char *buff = (char *)malloc(strlen(argv[1])+1);
   if (buff != NULL) {
     strcpy(buff, argv[1]);
     printf("argv[1] = %s.\n", buff);
    }
   else {
      /* Couldn't get the memory - recover */
    }
   return 0;
 }
```

Input Validation – Strings

common errors:

- Extracting characters from cin into a character array example:

```
int main(void) {
    char buf[12];

    cin >> buf;
    cout << "echo: " << buf << endl;
}
```

where is the problem?
what is cin?

```
istream& operator>> (istream& is, char* str);
```

the problem is: no truncation!

- Solution: (ios_base::width)

```
#include <iostream>

int main(void) {
char buf[12];

cin.width(12);
cin >> buf;
cout << "echo: " << buf << endl;
}
```

Input Validation – Strings

common errors:

How many errors do you find?

```
1. int main(int argc, char* argv[]) {
2.       char source[10];
3.       strcpy(source, "0123456789");
4.      char *dest = (char *)malloc(strlen(source))
5.      int i;
6.      for (i=1; i <= 11; i++) {
7.           dest[i] = source[i];
8.       }
9.       dest[i] = '\0';
10.      printf("dest = %s", dest);
11.}
```

1. line 2 and 3: source[10] is 10 bytes long but strcpy() copies 11 byte into it
2. line 4: malloc() does not allocate memory for terminating null byte
3. line 5: index i starts with 1 although array indices start with 0
4. line 5: i<=11 is counting one more than might be intended
5. line 8: terminating null will cause an out-of-bounds write

This type of errors is called “off-by-one” error.

Input Validation – Strings

common errors:

- what is the output of:

```
int main(int argc, char* argv[]) {
   char a[16];
   char b[16];
   char c[32];

   strcpy(a, "0123456789abcdef");
   strcpy(b, "0123456789abcdef");
   strcpy(c, a);
   strcat(c, b);
   printf("a = %s\n", a);
   return 0;
}
```

answer: ask the compiler it

depends on the management

of stack memory

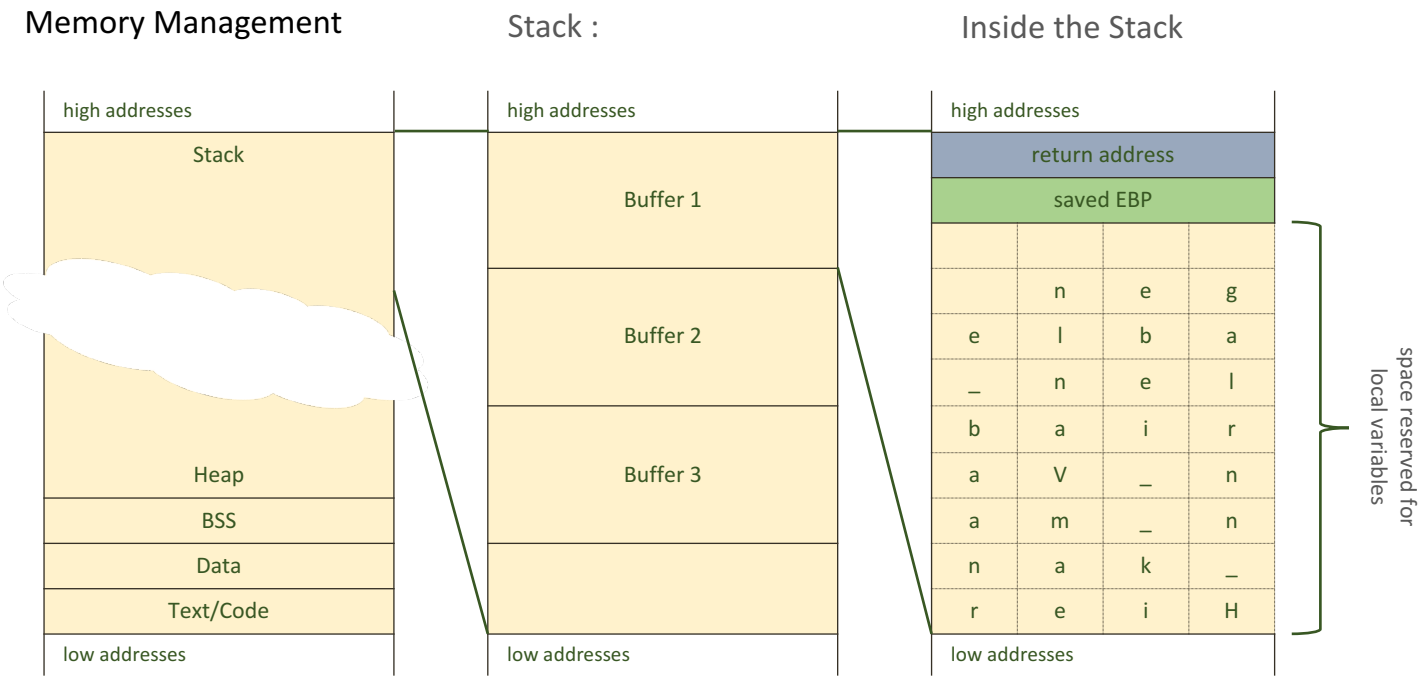
even worse: in depends on the content of allocated memory prior to the start of the program.

reason: null-termination mechanism, C-style-strings and string-literals are null-terminated

Input Validation – Strings

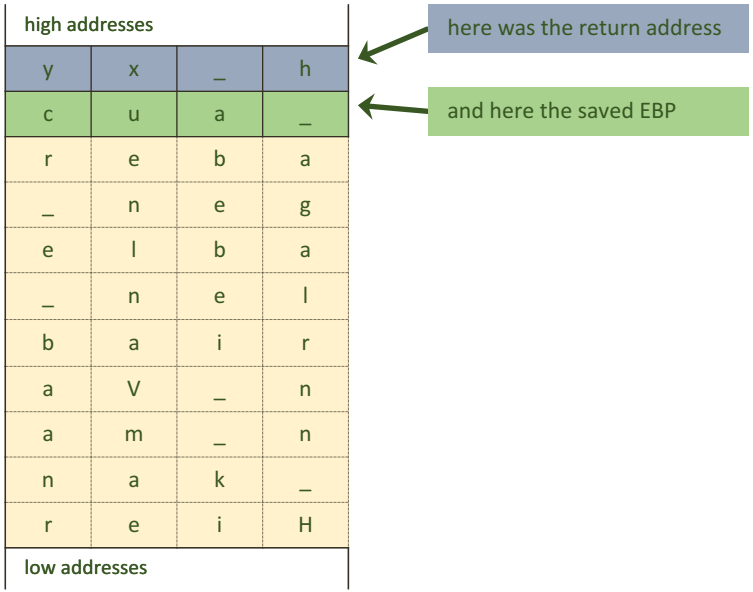
Vulnerabilities:

- What happens if you write more data to the stack than allowed?



Input Validation – Strings

- C and C++ simply don't care and continue to write to the memory
- consequence: the saved EBP and the return address will be overwritten



- in case the return address is overwritten with an invalid value, the program will crash (this is already a successful denial of service (DOS) attack)
- in case the return address is overwritten with a valid value, the program will continue from there. This opens up a range of opportunities for targeted manipulation of the program.

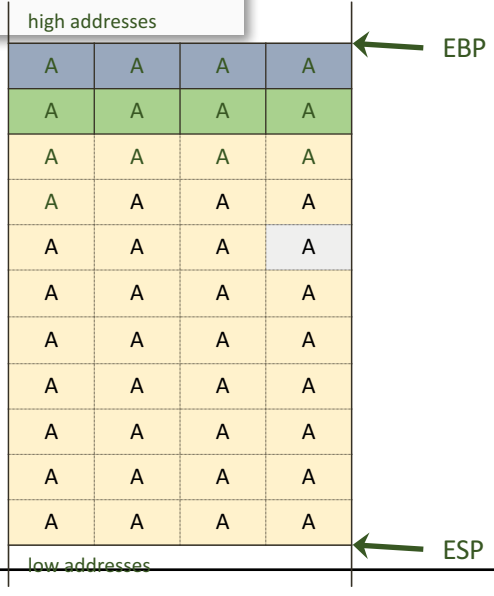
Input Validation – Strings

- Smashing the stack (example 1)

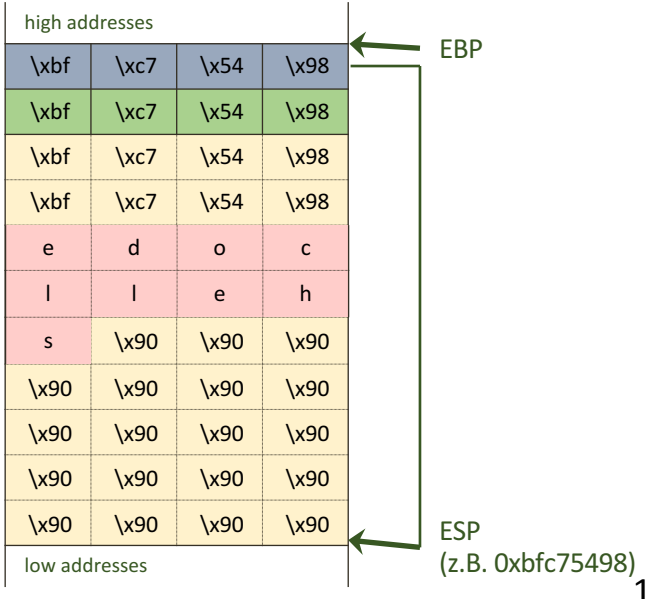
```
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv)
{
        char buffer [512];
        strcpy(buffer, argv[1]);
        printf("You entered:
%s\n",buffer);
        return 0;
}
```

1. identify vulnerability

   (by trying to crash the programm)



2. construct exploit
   a. NOPs (\x90)
      (cpu simply moves forward to the next input)
   b. executable code (e.g. /bin/sh)
   c. return address which leads into the area of NOPs



OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Input Validat – Strings

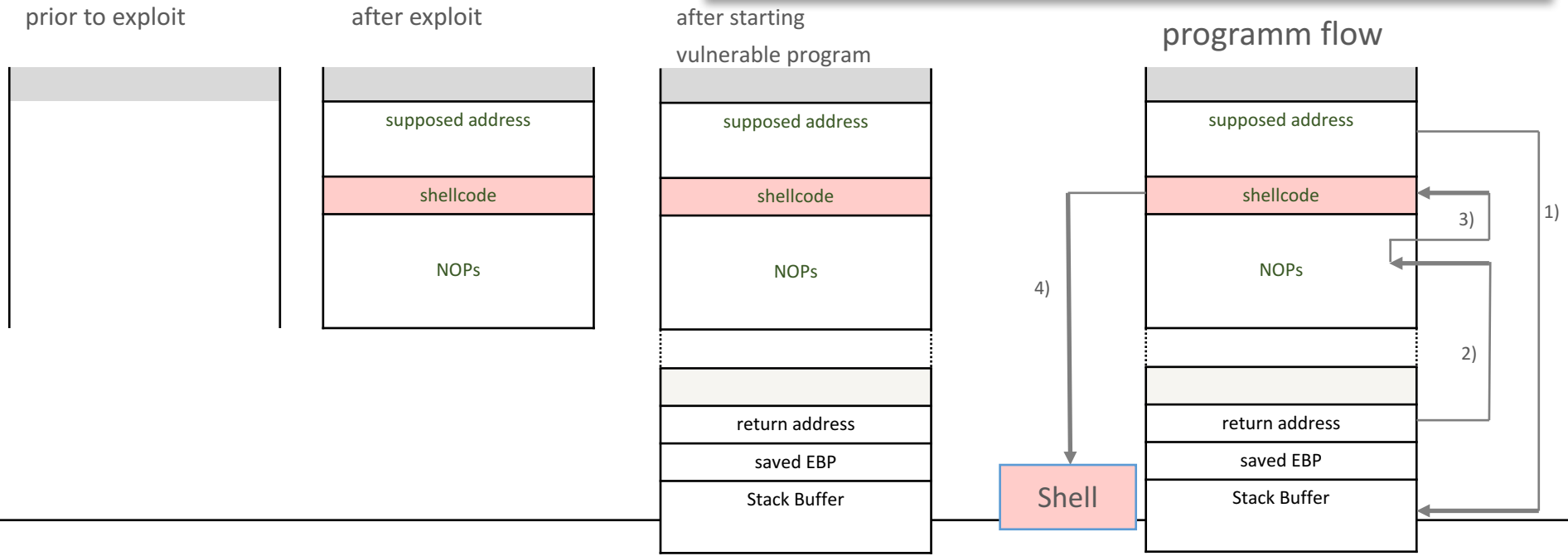
- Smashing the stack (example 2)
  - What is this program doing?
  - Is it vulnerable to DOS?
  - Will the already shown stack-smashing attack work?
  - Is there any option for a different type of attack?

```
05 void
06 geheim (void)
07 {
08      printf ("GEHEIM !!!\n");
09      exit (0);
10 }
11
12 void
13 oeffentlich (char *args)
14 {
15      char buff[12];
16
17      memset (buff, 'B', sizeof (buff));
18
19      strcpy (buff, args);
20      printf ("\nbuff: [%s] (%p)(%d)\n\n", &buff, buff, sizeof (buff));
21 }
22
23 int
24 main (int argc, char *argv[])
25 {
26      int uid;
27
28      uid = getuid ();
29
30      if (uid == 0)
31              geheim ();
32
33      if (argc > 1) {
34              printf ("geheim()      -> (%p)\n", geheim);
35              printf ("oeffentlich() -> (%p)\n", oeffentlich);
36
37              oeffentlich (argv[1]);
38
39              printf ("geheim() -> (%p)\n", geheim);
40              printf ("oeffentlich() -> (%p)\n", oeffentlich);
41      } else
42              printf ("Kein Argument!\n");
43
44      return 0;
45 }
```

Input Validation – Strings

- In the last example 12 byte is (most likely) not enough space to place your code in
- What to do now?

- idea: save malicious code anywhere and write address to EIP
- how to do that?
- lets try environment variables.



Input Validation – Strings

- Smashing the stack (example 3)
  - What is this program doing?
  - Is it vulnerable to DOS?
  - Will the already shown stack-smashing attacks work?
  - Is there any option for a different type of attack?
  - Answer: Off by one error in line10 allows potentially to overwrite last byte of EBP
  - Most likely it does not work but would it be a problem?

```
01 #include <stdio.h>
02
03 static void
04 function (char *args)
05 {
06 char buff[256];
07        int i;
08
09
10        for (i = 0; i <= 256; i++)
11        buff[i] = args[i];
12 }
13
14 int
15 main (int argc, char *argv[])
16 {
17        if (argc < 2) {
18        printf ("No Argument!\n");
19        exit (1);
20 }
21
22 function (argv[1]);
23
24        return 0;
25
26 }
```

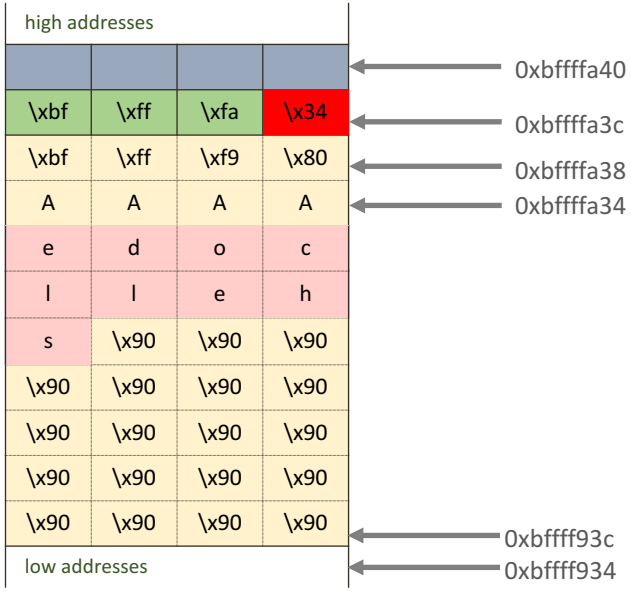
Input Validation – Strings

- Exploit: modification of last bit of saved ebp

Repetition: LEAVE and RET Function

LEAVE:
movl %ebp, %esp
popl %ebp

RET:
call func
movl $0, %eax **eip now points here**
addl $8, %esp



⇨ after LEAVE of func:
esp: 0xbffffa38 ebp:0xbffffa34

⇨ after RET of func:
esp: 0xbffffa40 ebp:0xbffffa34

⇨ after LEAVE in main:
esp: 0xbffffa38 ebp:0x41414141

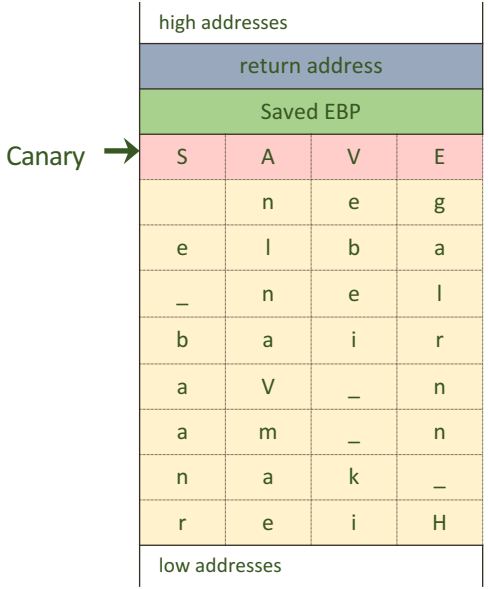
⇨ after RET in main:
eip: 0xbffff980

Modifying SFP will modify program flow

Attacks are possible e.g. OpenBSD FTP Daemon (2002)

Input Validation – Strings

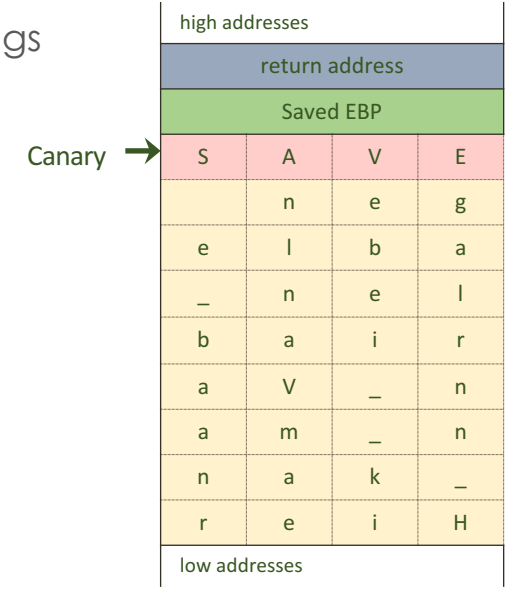
- Canaries
  - include a "canary" value which, when destroyed, shows that a buffer preceding it in memory has been overflowed
    - Terminator canaries (NULL, CR, LF, -1)
      problem: if canary is known, an attacker can try to work around it
    - Random canaries
      generated at program initialization and stored in a global variable
      (usually padded by unmapped pages, which makes it harder to read)
    - Random XOR canaries
      Random Canaries that are XOR scrambled using control data
      (change of control data a wrong canary)
  - Implementations: StackGuard (Unix), StackCookie (Win)
  - problem: DOS still working!



- Bounds checking
  - compiler-based technique that adds run-time bounds information for each allocated block of memory, and checks all pointers against those at run-time

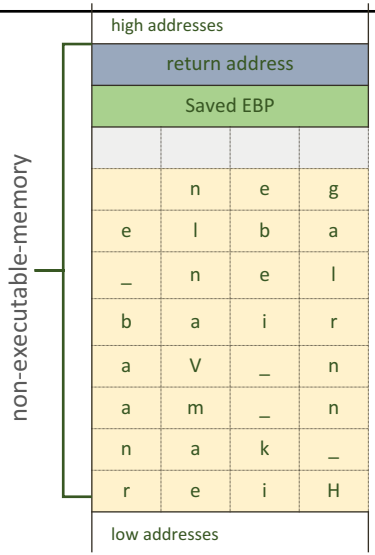
Input Validation – Strings

- AddressSanitizer (implemented in clang and gcc)
  - compiler-based tool to detect all kinds of memory-corruption bugs
  - `-fsanitize=address` oder `-fsanitize=kernel-address`
  - performance overhead(average):
    - 73% cpu time
    - 340% memory usage
  - Example features:
    - heap use after free
    - heap and Stack-buffer overflows



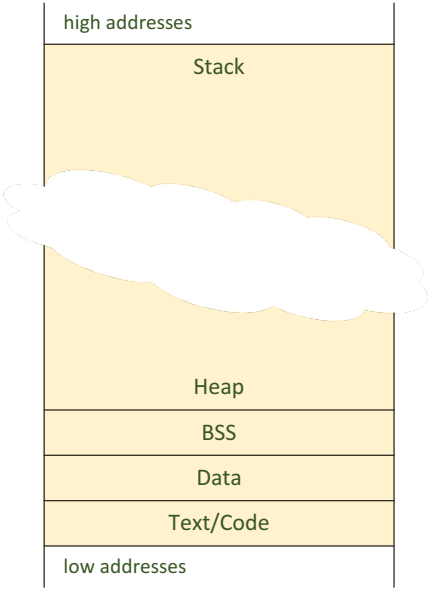
OS-based defense

- Non-Executable Stack
  - mark data in stack as non-executable (-z no-exec-stack) in compiler
- Address-Space-Layer-Randomization (ASLR)
  - randomize address spaces
  - guessing of appropriate return addresses made difficult
  - Implementations: Standard in Unix and Win Systems
- Sandboxes and compartment based OS extensions
  - potentially vulnerable components (like services, special applications etc.) are executed in protected areas.
  - See Qubes OS, Ethos or new Kaspersky Press release



Input Validation – Strings

- What about this?
- Any vulnerabilities?



- How about memory management on the heap?

```
1  int main(int argc, char *argv[]){
2      FILE *filed;
3      char *userinput = (char *) malloc(20);
4      char *outputfile = (char *) malloc(20);
5
6      strcpy(outputfile, "/mynotes");
7      strcpy(userinput, argv[1]);
8
9      printf("userinput @ %p: %s\n", userinput, userinput);
10     printf("outputfile @ %p: %s\n",outputfile, outputfile);
11
12
13     filed = fopen(outputfile, "a");
14     if(filed == NULL)
15     {
16         fprintf(stderr, "error opening file %s\n", outputfile);
17         exit(1);
18     }
19
20
21     fprintf(filed, "%s\n", userinput);
22     fclose(filed);
23     return 0;
24 }
```

Input Validation – Strings

- What is this program doing?
- Any vulnerabilities?

- What would be the output of:
  `./fstring-2 ABCD_%1\$x`
- and what about:
  `./fstring-2 ABCD_%50\$s`

```
1  int main (int argc, char *argv[]){
2      int     a = 1;
3      int     b = 2;
4      int     c = 3;
5      char    buff1[32];
6      char    buff2[] = "BBB";
7      int     d = 4;
8
9      memset (buff1, 'A', 32);
10     snprintf (buff1, sizeof (buff1), argv[1]);
11     buff1[sizeof (buff1) - 1] = '\0';
12     printf ("buff1: [%s] (%d)\n", &buff1,strlen
   (buff1));
13 }
```

Problem:

- `snprintf` is interpreting the format strings of the input

Consequence:

- read data from (normally not accessible) areas of the memory
- denial of service

Input Validation – Strings

- Any issues?

```c
#include <stdio.h>
#include <string.h>

int
main (int argc, char *argv[])
{
    char user[512];
    strncpy(user,argv[1],511);
    char outbuf[512];
    char buffer[512];
    sprintf(buffer,"ERR Wrong command: %.400s",user);
    sprintf(outbuf, buffer);
    return 0;
}
```

- What about

```
./fstring-5ABCD_%2000s
```

Buffer Overflow by buffer streching!

Input Validation – Strings

- What does this code do?

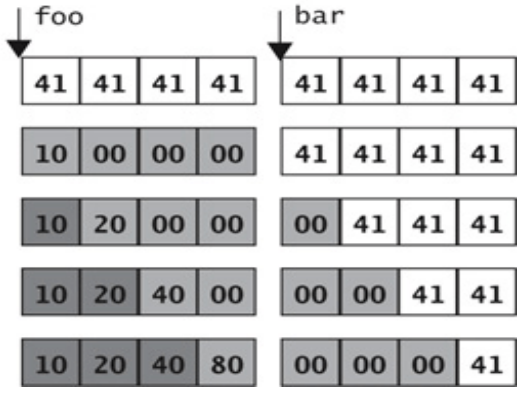
```
int i;
printf("hello%n\n", (int *)&i);
```

- And this?

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

Seems that it is possible to overwrite memory with that trick!

```
unsigned char foo[4];
memset(foo, '\x41', 4);

unsigned char bar[4];
memset(bar, '\x41', 4);

printf(
  "%16u%n", 1, &foo[0]);
printf(
  "%32u%n", 1, &foo[1]);
printf(
  "%64u%n", 1, &foo[2]);
printf(
  "%128u%n", 1, &foo[3]);
```



Input Validation – Strings

Overwrite arbitrary address with arbitrary value

- Lets look at the internal flow:

```
snprintf (buff1, sizeof (buff1), \x90\xf3\xff\xbf_%x_%x_%x_%n)
```

Format Strings:

- %x: Unsigned hexadecimal integer
- %n: Nothing printed.

The corresponding argument must be a pointer to a signed int.

The number of characters written so far is stored in the pointed location.

%n is overwriting the address at the first four bytes of buff1 (0xbffff390) with the length of buff1 (22)

- Even better

```
snprintf (buff1, sizeof (buff1), \x90\xf3\xff\xbf_%x_%x_%.10x_%n)
```

.10x is setting the length of the output and therefore the length of buff1

a Overwrite arbitrary address with arbitrary value!!!

Input Validation – Strings

Conclusion:

- Never use:
  - `printf (string);`
  - `fprintf (stderr, string);`
  - `snprintf (puffer, sizeof (puffer), string)`
- Better:
  - `printf ("%s", string);`
  - `fprintf (stderr, "%s", string);`
  - `snprintf (puffer, sizeof (puffer), "%s", string);`

Consider Compiler Warning options:

```
-fsyntax-only -pedantic -pedantic-errors -w -Wextra -Wall -Waddress -Waggregate-return -Warray-bounds -Wno-attributes -Wno-builtin-macro-redefined -Wc++-compat -Wc++0x-compat -Wcast-align -Wcast-qual -Wchar-subscripts -Wclobbered -Wcomment -Wconversion -Wcoverage-mismatch -Wno-deprecated -Wno-deprecated-declarations -Wdisabled-optimization -Wno-div-by-zero -Wempty-body -Wenum-compare -Wno-endif-labels -Werror -Werror=* -Wfatal-errors -Wfloat-equal -Wformat -Wformat=2 -Wno-format-contains-nul -Wno-format-extra-args -Wformat-nonliteral -Wformat-security -Wformat-y2k -Wframe-larger-than=len -Wignored-qualifiers -Wimplicit -Wimplicit-function-declaration -Wimplicit-int -Winit-self -Winline -Wno-int-to-pointer-cast -Wno-invalid-offsetof -Winvalid-pch -Wlarger-than=len -Wunsafe-loop-optimizations -Wlogical-op -Wlong-long -Wmain -Wmissing-braces -Wmissing-field-initializers -Wmissing-format-attribute -Wmissing-include-dirs -Wmissing-noreturn -Wno-mudflap -Wno-multichar -Wnonnull -Wno-overflow -Woverlength-strings -Wpacked -Wpacked-bitfield-compat -Wpadded -Wparentheses -Wpedantic-ms-format -Wno-pedantic-ms-format -Wpointer-arith -Wno-pointer-to-int-cast -Wredundant-decls -Wreturn-type -Wsequence-point -Wshadow -Wsign-compare -Wsign-conversion -Wstack-protector -Wstrict-aliasing -Wstrict-aliasing=n -Wstrict-overflow -Wstrict-overflow=n -Wswitch -Wswitch-default -Wswitch-enum -Wsync-nand -Wsystem-headers -Wtrigraphs -Wtype-limits -Wundef -Wuninitialized -Wunknown-pragmas -Wno-pragmas -Wunreachable-code -Wunused -Wunused-function -Wunused-label -Wunused-parameter -Wunused-value -Wunused-variable -Wunused-but-set-parameter -Wunused-but-set-variable -Wvariadic-macros -Wvla -Wvolatile-register-var -Wwrite-strings
```

```c
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
    int     a = 1;
    int     b = 2;
    int     c = 3;
    char    buff1[32];
    char    buff2[] = "BBB";
    int     d = 4;

    memset (buff1, 'A', 32);
    snprintf (buff1, sizeof (buff1),"%s",argv[1]);
    buff1[sizeof (buff1) - 1] = '\0';
    printf ("buff1: [%s] (%d)\n", &buff1,strlen (buff1));
}
```

Input Validation – Strings

Famous examples:

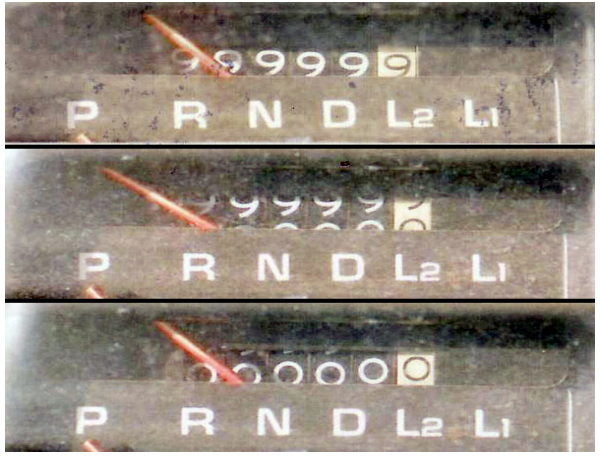
- format string in Perl program
- bad call to syslog function
- bad call to syslog function
- NNTP server responses
- specifiers in a .bmp filename
- malicious internationalization messages

Input Validation – Integers

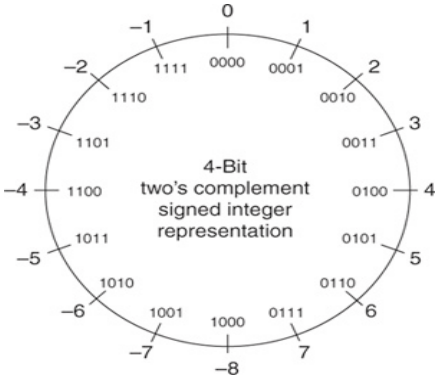
Regular errors with Integers:

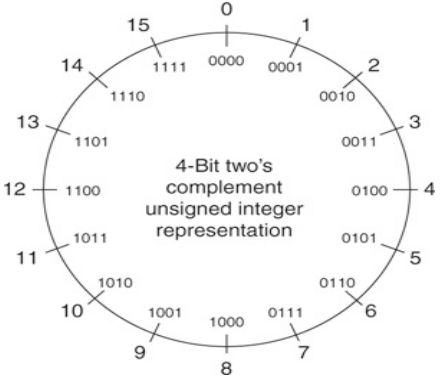
- Integer Overflow → e.g. encryption uses very large integers
- Sign Errors
- Truncation Error

try to know how large the integers can and will possibly get in your programs




```c
int main (){
    int i;
    unsigned int j;

    i = 2147483647;
    i++;
    printf("i = %d\n", i);

    j = 429967295;
    j++;
    printf("j = %u\n", j);

    i = -2147483647;
    i--;
    printf("i = %d\n", i);

    j = 0;
    j--;
    printf("j = %u\n", j);
    i = -3;
    unsigned short u;

    u = i;
    printf("u = %hu\n", u);

    unsigned short int us = 32768;
    short int is;

    is = us;
    printf("is = %d\n", is);

    us = 65535;
    is = us;
    printf("is = %d\n", is);
}
```



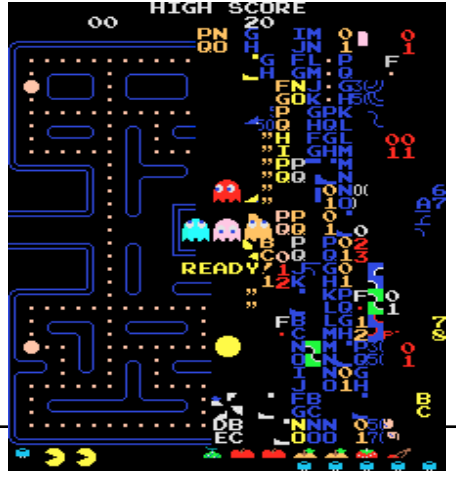


Input Validation – Integers

Corresponding Vulnerabilities:

- Integer Overflow
  - example (2000):
    real world integer overflow in jpeg comment fields
    - if comment field size is 1 then the comment length calculates to 1-2=0xffffffff
    - memory allocation goes terribly wrong
  - example (2002):
    computation of the memory region size by calloc
    - http://cert.uni-stuttgart.de/ticker/advisories/calloc.html
- Similar Examples for sign and truncation errors
  - http://www.kb.cert.org/vuls/id/540517
  - … (ask google)

```
void getComment(unsigned int len, char *src) {
    unsigned int size;

    size = len - 2;
    char *comment = (char *)malloc(size + 1);
    memcpy(comment, src, size);
    return;
}

int _tmain(int argc, _TCHAR* argv[]) {
    getComment(1, "Comment ");
    return 0;
```



Input Validation – Integers

Corresponding Vulnerabilities:

- Any Vulnerability here?

What happens if pos=-4?

negative position in table → write into memory addresses you're not allowed to

```
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;
}
```

pos value is passed as an argument, value would be written to a location pos x 4 bytes before the start of the actual buffer!

Better:

```
int *table = NULL;

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

Input Validation – Integers

Mitigation Strategies:

- Range Checking
- Strong Typing
- Compiler Options (e.g. –ftrapv on gcc)
- Arbitrary Precision Arithmetic
  - GNU Multiple Precision Arithmetic Library (GMP)
  - Java BigInteger

Input Validation Additional Topics

- Validating Email-Addresses
  - check against RFC 822
  - default deny

- Cross Site Scripting
  - Refuse to accept anything that looks as if it may be HTML
  - Escape the special characters that enable a browser to interpret data as HTML
  - default deny

- SQL-Injection
  - Restrict user input to the smallest character set possible and refuse any input that contains characters outside of that set
  - Escape character that have special significance to SQL command processors
  - default deny

Default Deny!!!

thanks for your interest

to be continued