Arithmetic Overflow

In mathematics: $a+b>a$ and $a-b<a$ for $b>0$

- Such obvious facts are no longer true for binary represented integers

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

int main(int argc, char* argv[]) {

    unsigned int u1 = UINT_MAX;
    u1 ++;
    printf(“u1 = %u\n", u1);

    unsigned int u2 = 0;
    u2 --;
    printf(“u2 = %u\n", u2);

    signed int s1 = INT_MAX;
    s1 ++;
    printf(“s1 = %d\n", s1);

    signed int s2 = INT_MIN;
    s2 --;
    printf(“s2 = %d\n", s2);
}
```

⇨ 4,294,967,295

⇨ 0

⇨ 4,294,967,295

⇨ 2,147,483,647

⇨ -2,147,483,648

⇨ -2,147,483,648

⇨ 2,147,483,647

Example 1: Bypass Length Checking

Incorrect length checking could lead to integer overflows, and then buffer overflow.

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

char buf[128];
void combine(char *s1, unsigned int len1, char *s2, unsigned int len2) {
  if (len1 + len2 + 1 <= sizeof(buf)) {
   strncpy(buf, s1, len1);
   strncat(buf, s2, len2);
  }
}

int main(int argc, char* argv[]) {
    unsigned int len1 = 10;
    unsigned int len2 = UINT_MAX;
    char *s1 = (char *)malloc(len1 * sizeof(char));
    char *s2 = (char *)malloc(len2 * sizeof(char));
    combine(s1, len1, s2, len2);
}
```

Buffer Overflow!

`len1 + len2 + 1 = 10 < 128`
`strncpy` and `strncat` will be executed.

Widthness Overflow

A bad type conversion can cause widthness overflows

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

int main(int argc, char* argv[]) {

    unsigned int l = 0xdeabeef;
    printf(“l = 0x%u\n", l);

    unsigned short s = l;
    printf(“s = 0x%u\n", s);

    unsigned char c = l;
    printf(“c = 0x%u\n", c);

}
```

⇨ 0xdeadbeef

⇨ 0xbeef

⇨ 0xef

Example 2: Truncation Errors

Incorrect type conversion could lead to integer overflows, and then buffer overflow.

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

int func(char *name, unsigned long cbBuf) {
    unsigned int bufSize = cbBuf;
    char *buf = (char *)malloc(bufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        free(buf);
        return 0;
    }
}

int main(int argc, char* argv[]) {
    unsigned long len = 0x10000ffff;
    char *name = (char *)malloc(len * sizeof(char));
    func(name, len);
}
```

`bufSize = 0xffff`

Buffer Overflow!

How to Fix Integer Overflow Vulnerability

Be more careful about all the possible consequences of vulnerable operations.

Better length checking

```
if (len1 + len2 + 1 <= sizeof(buf))
```

⇨

```
if (len1 <= sizeof(buf) &&
    len2 <= sizeof(buf) &&
    (len1 + len2 + 1 <= sizeof(buf)))
```

Safe type conversion:

- Widening conversion: convert from a type of smaller size to that of larger size.

Outline

- Format String Vulnerabilities
- Integer Overflow Vulnerabilities
- **Scripting Vulnerabilities**

Scripting Vulnerabilities

Scripting languages

- Construct commands (scripts) from predefined code fragments and user input at runtime
- Script is then passed to another software component where it is executed.
- It is viewed as a domain-specific language for a particular environment.
- It is referred to as very high-level programming languages
- Example:
  - Bash, PowerShell, Perl, PHP, Python, Tcl, Safe-Tcl, JavaScript

Vulnerabilities

- An attacker can hide additional commands in the user input.
- The system will execute the malicious command without any awareness

Example 1: Command Injection

Consider a server running the following command

- `system`: takes a string as input, spawns shell, and executes the string as command in the shell.

```
void display_file(char* filename) {
  char cmd[512];
  snprintf(cmd, sizeof(cmd), “cat %s”, filename);
  system(cmd);
}
```

Normal case:

- A client sets `filename`=hello.txt

```
cat hello.txt
```

Compromised Input:

- The attacker sets `filename` = hello.txt; rm –rf /
- The command becomes:

```
cat hello.txt; rm -rf /
```

- After displaying file, all files the script has permission to delete are deleted!