Software Vulnerabilities

Introduction

- Who I am
- Who this talk is for

Roadmap

- Buffer overflows – stack and heap.
- Format String bugs
- Integer problems
- Memory problems
- Race conditions

Roadmap

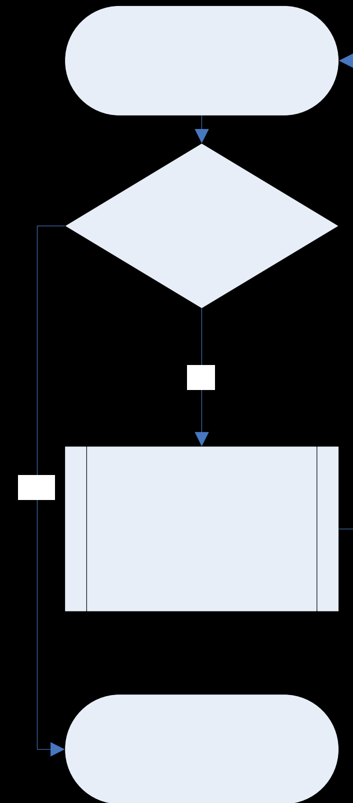
EXAMPLES?

or…

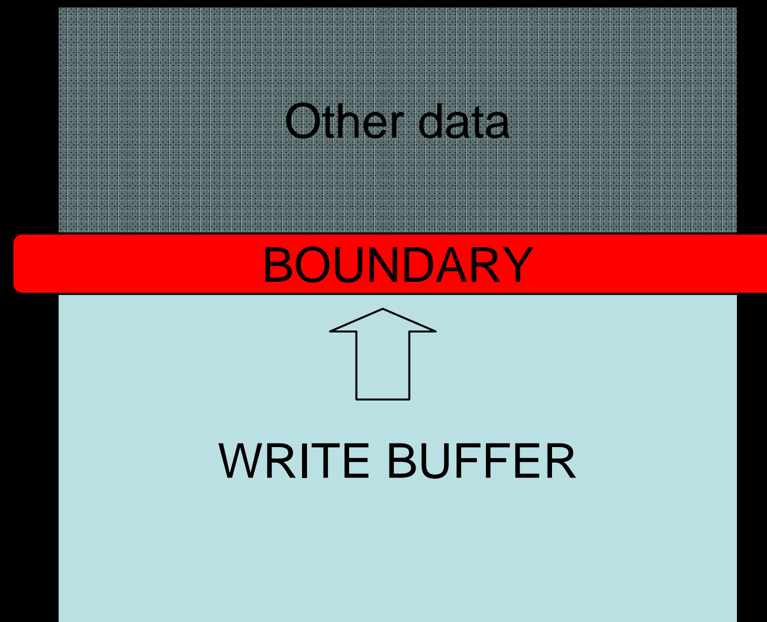
- Local vs Remote exploitation
- Post exploitation payloads
- Protection

The overflow

```
for(
  i = 0;
  user_buf[i] != ‘\0’;
  store_buf[i] = user_buf[i++]
);
```



How the attack works



Stack based overflow

Saved frame pointer:

```
call vuln_function

push %ebp
movl %esp, %ebp
[overflow]
movl %ebp, %esp
popl %ebp
```

Saved variables:

```
void (*fn)(void) = (void)&time;
char overflow_me[4];
[overflow]
fn();
```

Heap based overflow

.bss overwrite:

```
char p1[100], p2[100];

int main(void)
{
        overflow(p1);
}
```

Control structure overwrite:

```
void fn(void)
{
        char *p1 = malloc(100);
        char *p2 = malloc(100);
        overflow(p1);
        [free(), malloc(), whatever()]
        return;
}
```

What is the trend?

- You sequentially overwrite data beyond(/before) an allocated memory space.
- You rely on the position of the buffer to determine the way to attack (heap, stack).
- The attack is nothing more then modifying data at particular offsets to values that will leverage exploitation.

Format String Bugs

```
char *p = user_supplied_format_string;
printf(p);
```

How the attack works

printf

user supplied format string

AAAABBBB%08x%08x%08x%08x…

DPA - %12 etc

- `overwrite_addr1 = stack_offset(user_buffer);`
- `overwrite_addr2 = stack_offset(user_buffer) + 4;`

%hn

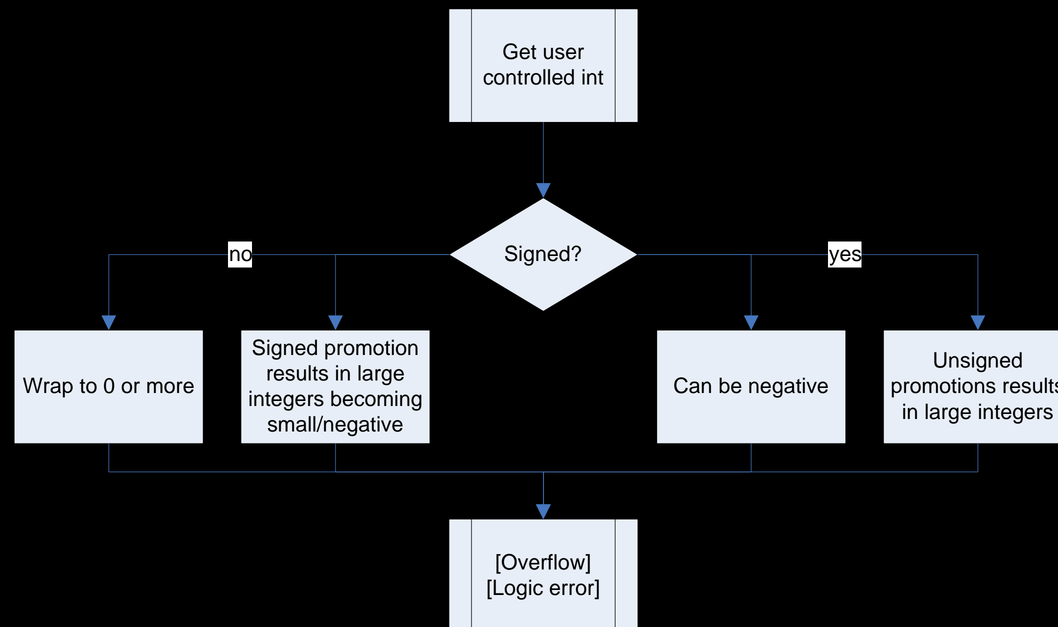
- **Write 1 = low_word(shellcode_addr) - bytes(already_output);**

- **Write 2 = high_word(shellcode_addr) - bytes(already_output);**

What is the trend?

- User input describes the format string used to describe argument types to format functions.
- The condition “write anything anywhere” allows an attacker to write arbitrary values to any memory location of their choice.
- syslog(), printf(), sprinf(), vfprintf(), hand made “format string” functions providing arbitrary read||write functionality, etc are vulnerable to this style of attack.

Integer problems



Arithmetic overflow

```
unsigned len = user_assigned_int32();
char *buf = malloc(len + 1);
memcpy(buf, user_supplied_input, len);
```

Signedness Bugs

```
signed len = user_assigned_int32();
if(len > sizeof(buffer)) return;
memcpy(buffer, user_supplied_input, len);
```

What is the trend?

- Arithmetic operations on integers that an attacker can control are dangerous.
- Promotions can happen to signed/unsigned values which result in logic errors.
- When these appear in loops or memory access/allocation an attacker is usually able to leverage an overflow or “write anything anywhere” condition.

Memory problems

- Memory leak - do not free() allocated memory.
- Uninitialized variables - Do not initialize variables before use.
- Uninitialized variables - Use an already free()'d memory block.

Memory leak

```
void f(void)
{
        char *s = malloc(100);
        if(!s) return -1;
        […]
        return 0;
}

int main(void)
{
        char *p;
        while(!f());
        p = malloc(size);
        write_to(p);
}
```

Un-initialized variables

```
unsigned int we_are;

if(we_are == 31337)
{
        root_us();
}
```

Un-initialized memory cont.

```
for(chnk = head; chnk; chnk = chnk->next)
{
   memset(chnk, 0x00, chnk->len);
   free(chnk);
}
```

What is the trend?

- Memory leaks can lead to some exploitable conditions when no further error checking is done on allocations.
- Sensitive data may be re-used or leaked somehow.
- Not initializing data can result in attacker-supplied data being used instead - leading to possibly exploitable scenarios.
- Referencing memory that has been free'd can lead to "un-expected" results.

Race Conditions

File handling race.

Signal handling race.

Threaded race.

File Handling Race Condition

```
struct stat st;
FILE *fp;

if(stat(argv[1], &st) < 0) {
      perror("stat");
      exit();
}

if(st.st_uid != getuid()) {
      fprintf(stderr, "you must be the owner of '%s'\n", argv[1]);
      exit();
}

if(!S_ISREG(st.st_mode)) {
      fprintf(stderr, "%s is not a normal file!\n", argv[1]);
      exit();
}

if((fp = fopen(argv[1], "w")) == NULL) {
      fprintf(stderr, "Failed to open %s\n", argv[1]);
      exit();
}

fprintf(fp, "%s\n", argv[2]);
```

Signal Handling Race Conditions

```
void sh(int dummy) {
  syslog(LOG_NOTICE,"%s\n",what);
  free(global2);
  free(global1);
  sleep(10);
  exit(0);
}

int main(int argc,char* argv[]) {
  what=argv[1];
  global1=strdup(argv[2]);
  global2=malloc(340);
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
```

Thread race conditions

```
unsigned long v;

void * thread1(void *p)
{
     v  = 1;
     v += 2;
}

void * thread2(void *p)
{
     v += 100;
}
```

What is the trend?

- It is just that – a “race” – between files, threads, and signals.
- Any non-atomic operation may be exploitable.

EXAMPLES?

Local vs Remote

- Environment challenges
- Blind exploitation
- Service identification

Post exploitation

- Architecture spanning shellcode
- Platform independent shellcode

Protections

- Address space layout randomization (ASLR)
- Control structure sanitization
- No Execute

mercy@felinemenace.org