Secure Programming

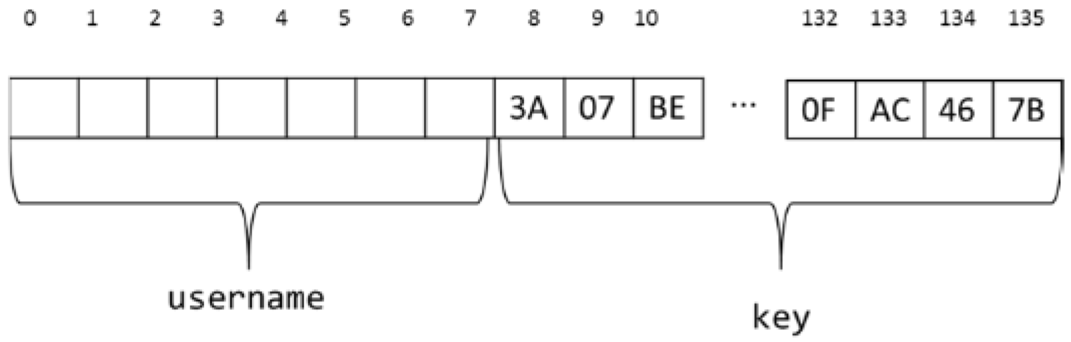
CPE 545

Buffer overflow

- Buffer Overflow
  - Is an anomaly that occurs when a program reads or writes data beyond the boundary of a buffer.
  - Can be exploited to overwrite memory adjacent to a buffer, thus corrupting memory that may contain control flow information, passwords, cryptographic key, etc..

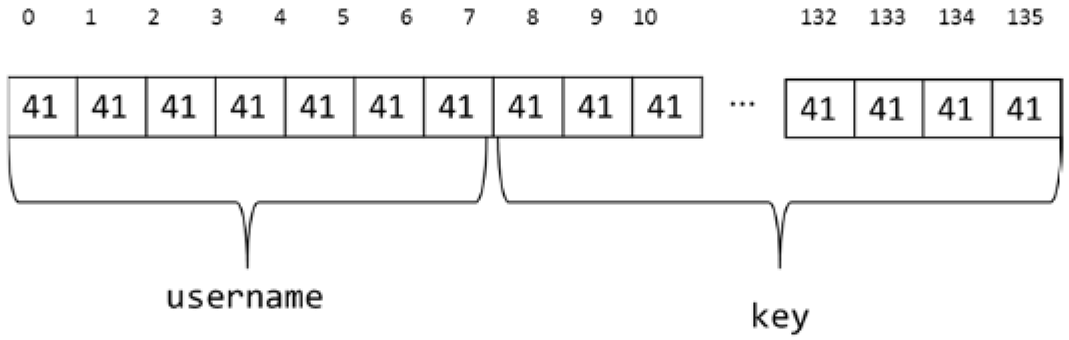
Buffer overflow

- 8 bytes username followed by cryptographic key stored in memory

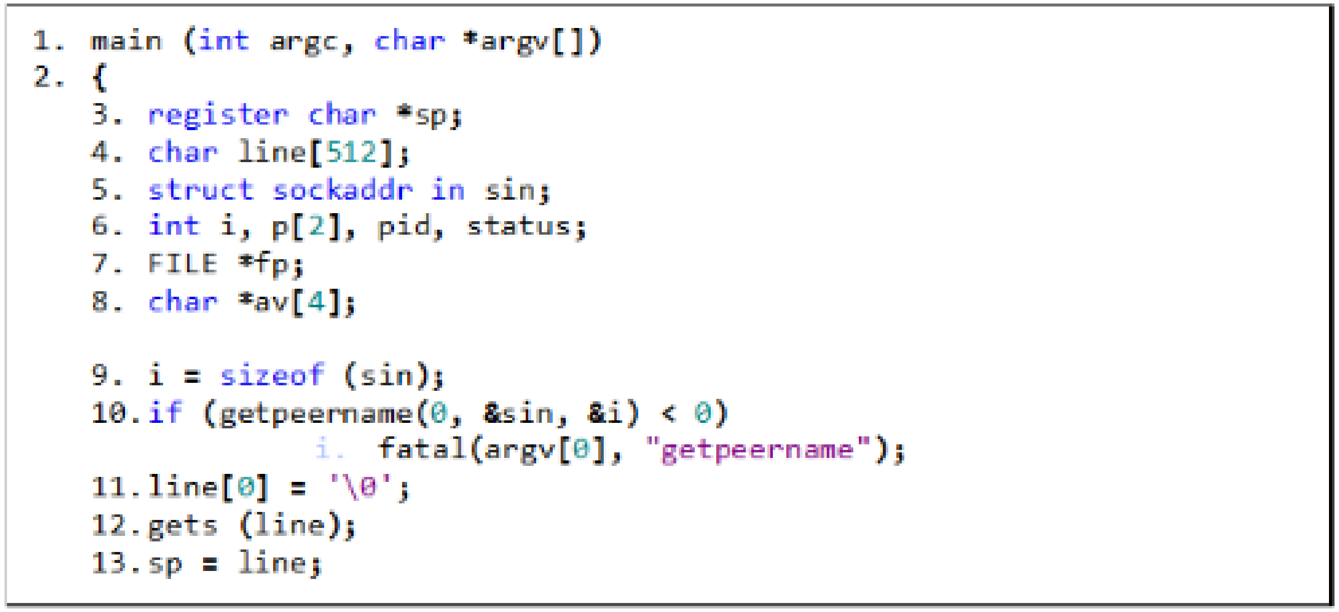


Buffer overflow

- How can buffer overflow be exploited?
  - A function copies username onto stack using the length provided by user. If length is greater than 8 bytes, an attacker can exploit this vulnerability by providing input to overwrite the key.



Buffer overflow

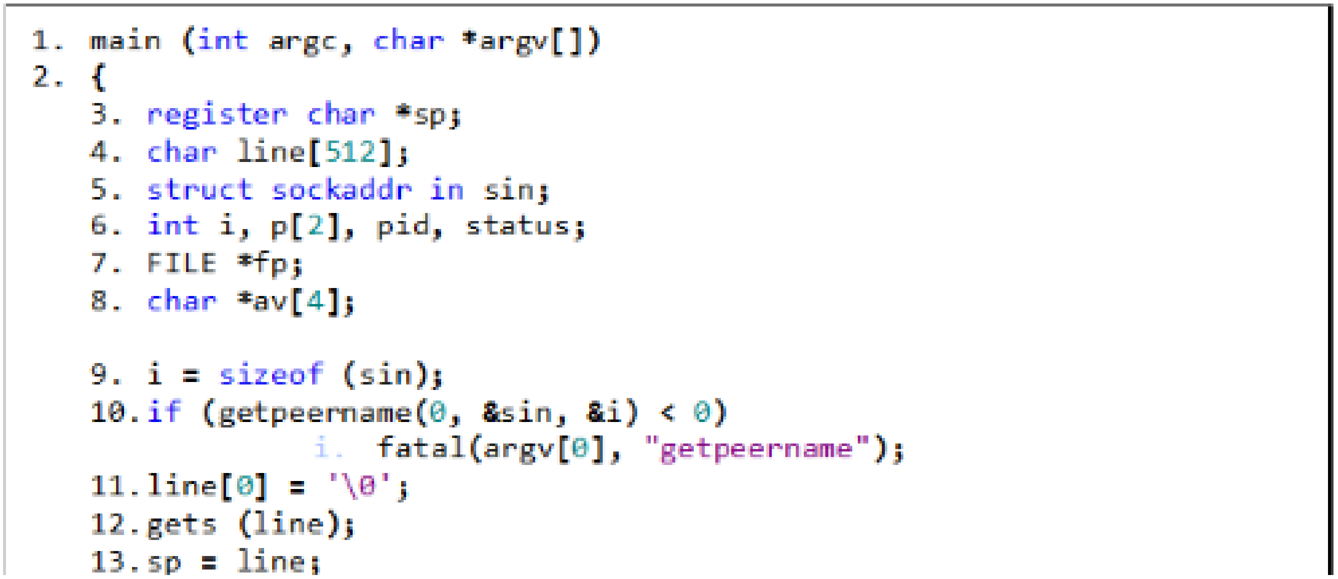


```
main (int argc, char *argv[])
{
    register char *sp;
    char line[512];
    struct sockaddr in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
            fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets (line);
    sp = line;
```

Can you spot the vulnerability?

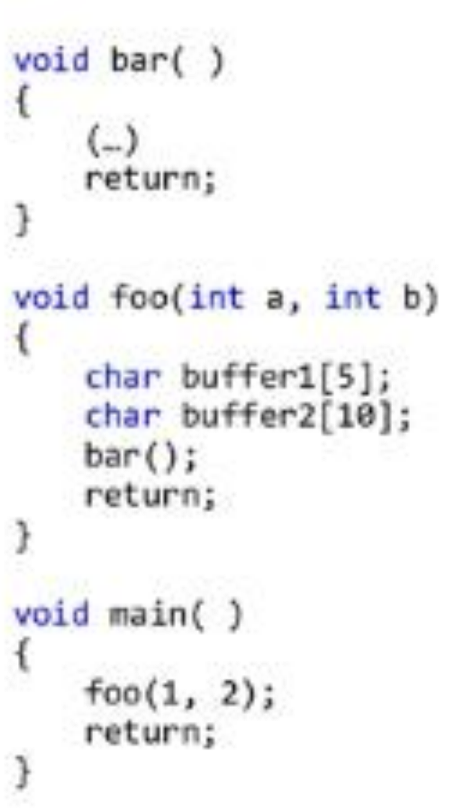
Buffer overflow

- gets (line), does not check any length checking, so a user input longer than 512 bytes can cause buffer over overflow

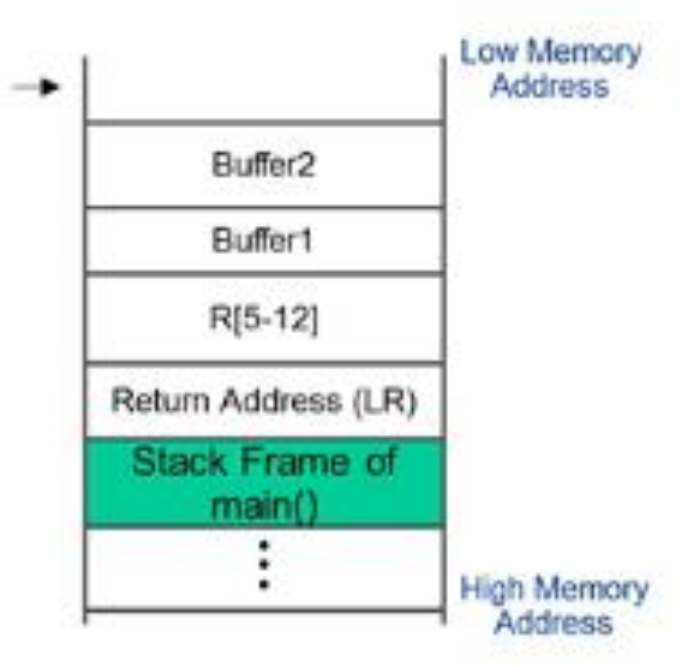


```c
main (int argc, char *argv[])
{
    register char *sp;
    char line[512];
    struct sockaddr in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
              fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets (line);
    sp = line;
```

Stack-based buffer overflow

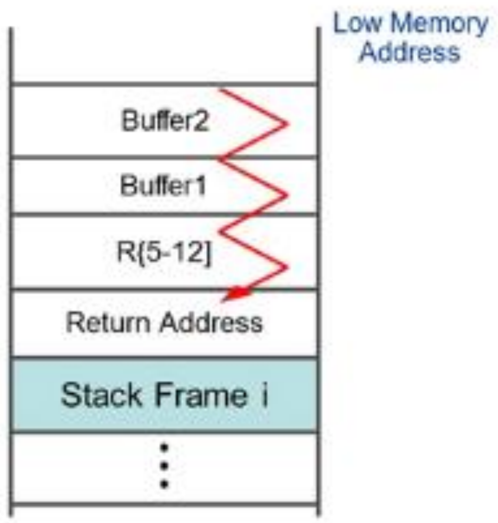


```
void bar( )
{
    (…)
    return;
}

void foo(int a, int b)
{
    char buffer1[5];
    char buffer2[10];
    bar();
    return;
}

void main( )
{
    foo(1, 2);
    return;
}
```



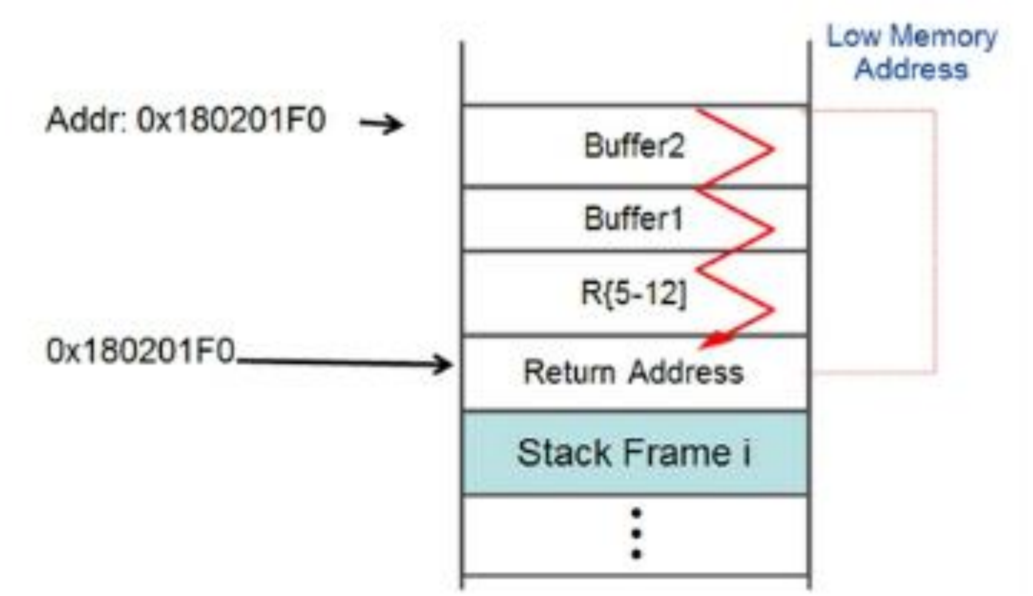
Stack-based buffer overflow

- The input provided by attacker would corrupt the contents of buffer 1[] and overwrite the adjacent saved variable registers, the saved LR that stored the returned address.



Stack-based buffer overflow

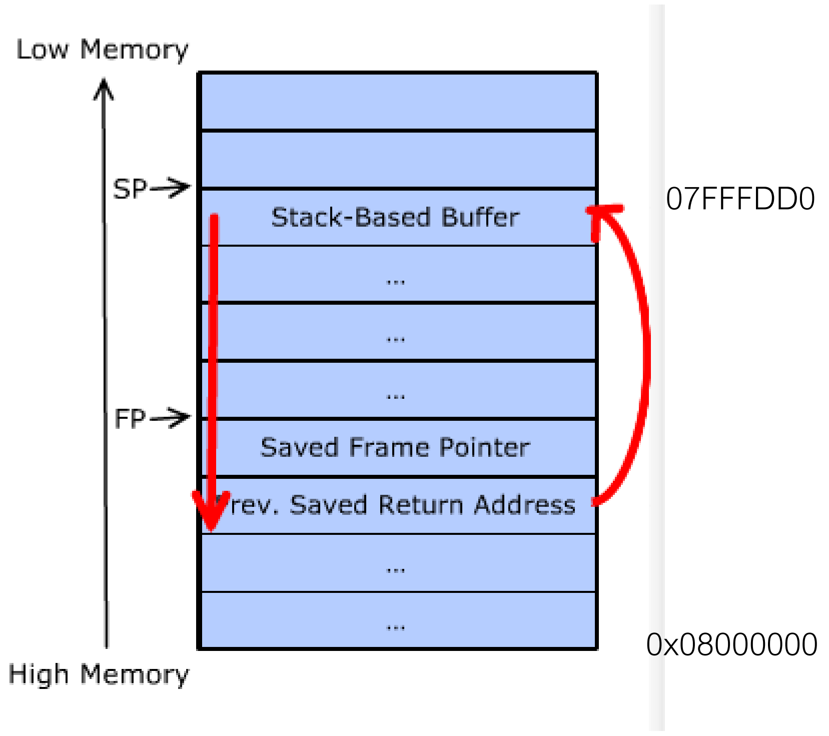
- Attacker may embed malicious binary code inside her input and overwrite the saved returned address on stack with the address of her malicious code in buffer2.
- When function foo() returns, it jumps to the attacker controlled address.

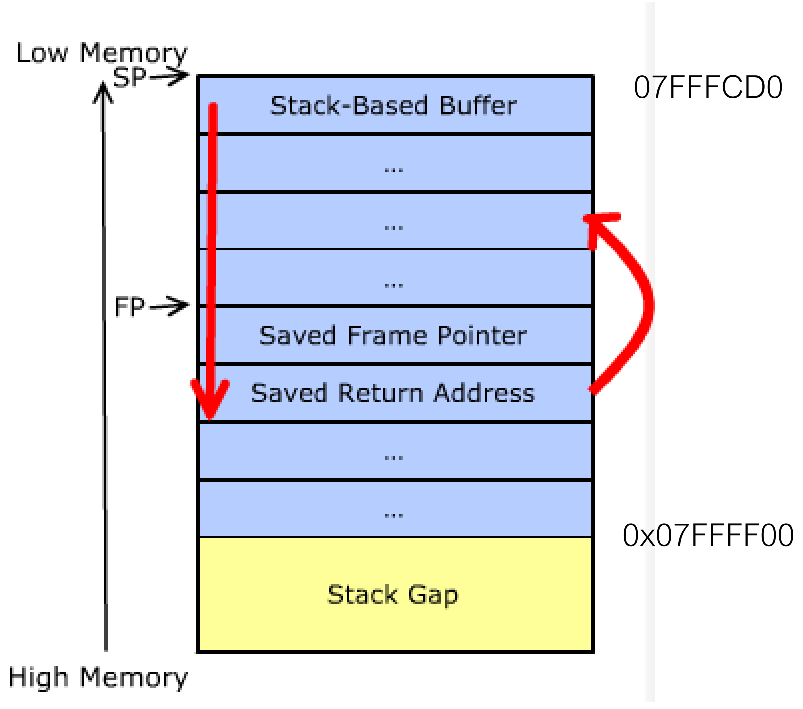


Stack Gap

- Stack gap is a mitigation to address execution of malicious code in stack-based buffer overflow, by adding a random number to the base of stack.
  - Stack will be located at different location with each execution of program. This changes the address of the attacker controlled buffer, preventing him from reliably jumping to his code.

Stack Gap

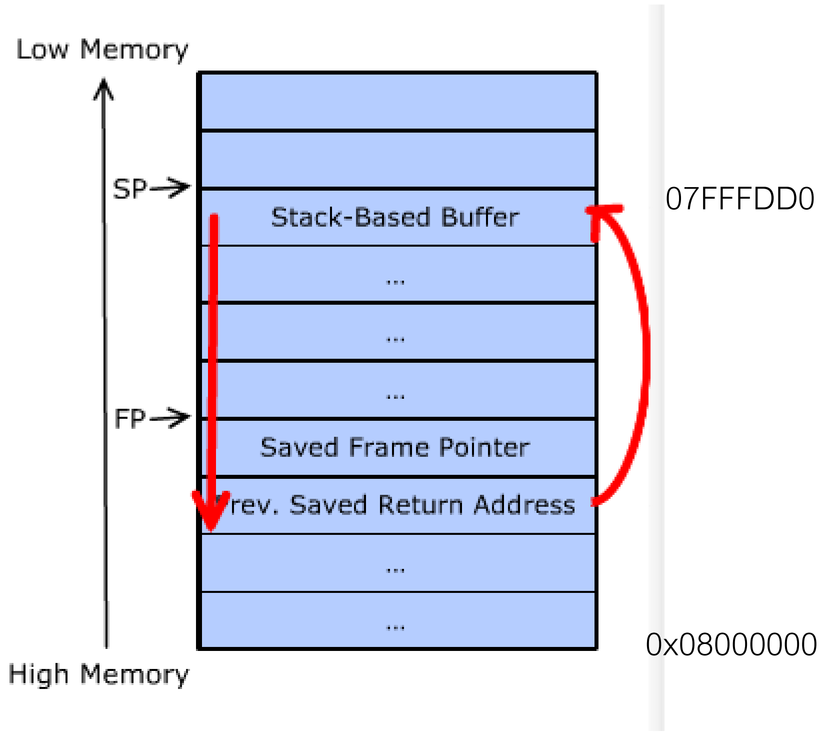


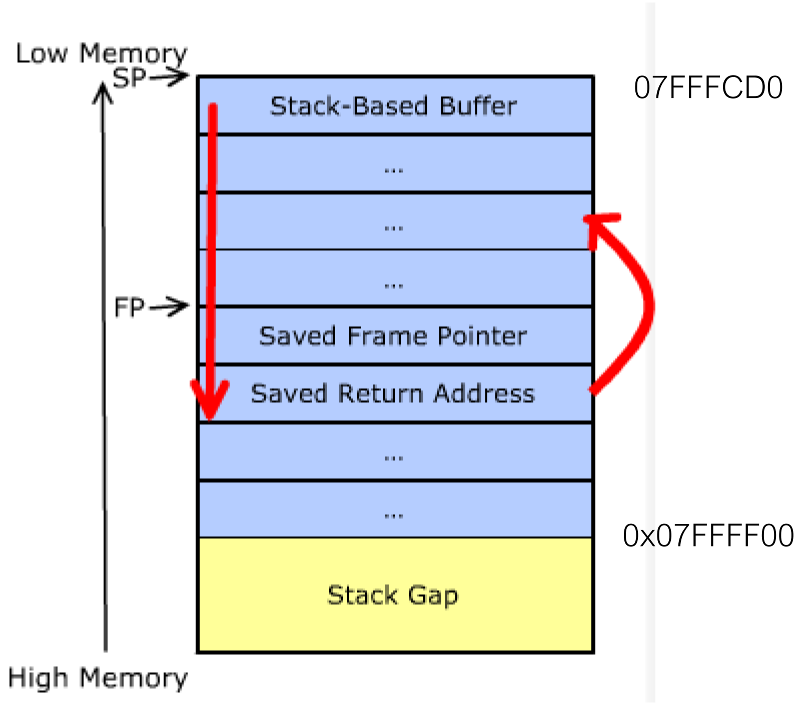


If the default stack base is 0x0800000 and the buffer address is at 0x07FFFDDo

With stack gap 0f 0x100, the new stack base will be 0x07FFFF00 and buffer will begin at 0x07FFFCD0

Stack Gap





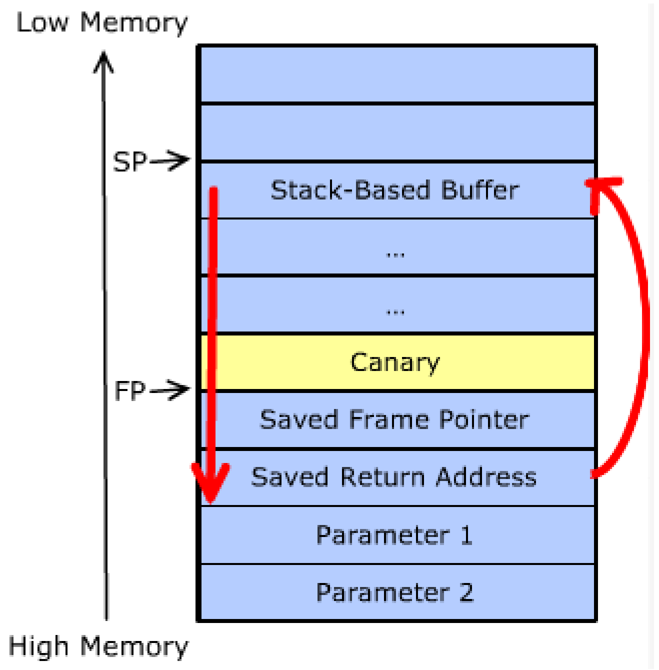
Can be circumvented by brute force approach, trying different offset until the attack succeeds

Stack Canaries

- Canary is placed before the return address on the stack and is initialized with a known value
- Before jumping to address pointed to by returned address, the value of canary is compared to the known value.
- If the canary value is the same, the program continues execution from the saved return address.
- Otherwise, if the canary value is changed, this indicates the return address is overwritten and program aborts

Stack Canaries

- Terminator Canaries are composed of sequences of bytes that terminates writing to a buffer, e.g. 0x000a0dff (null, line feed, carriage return)
- Random canaries are random sequence of bytes and harder to spoof.



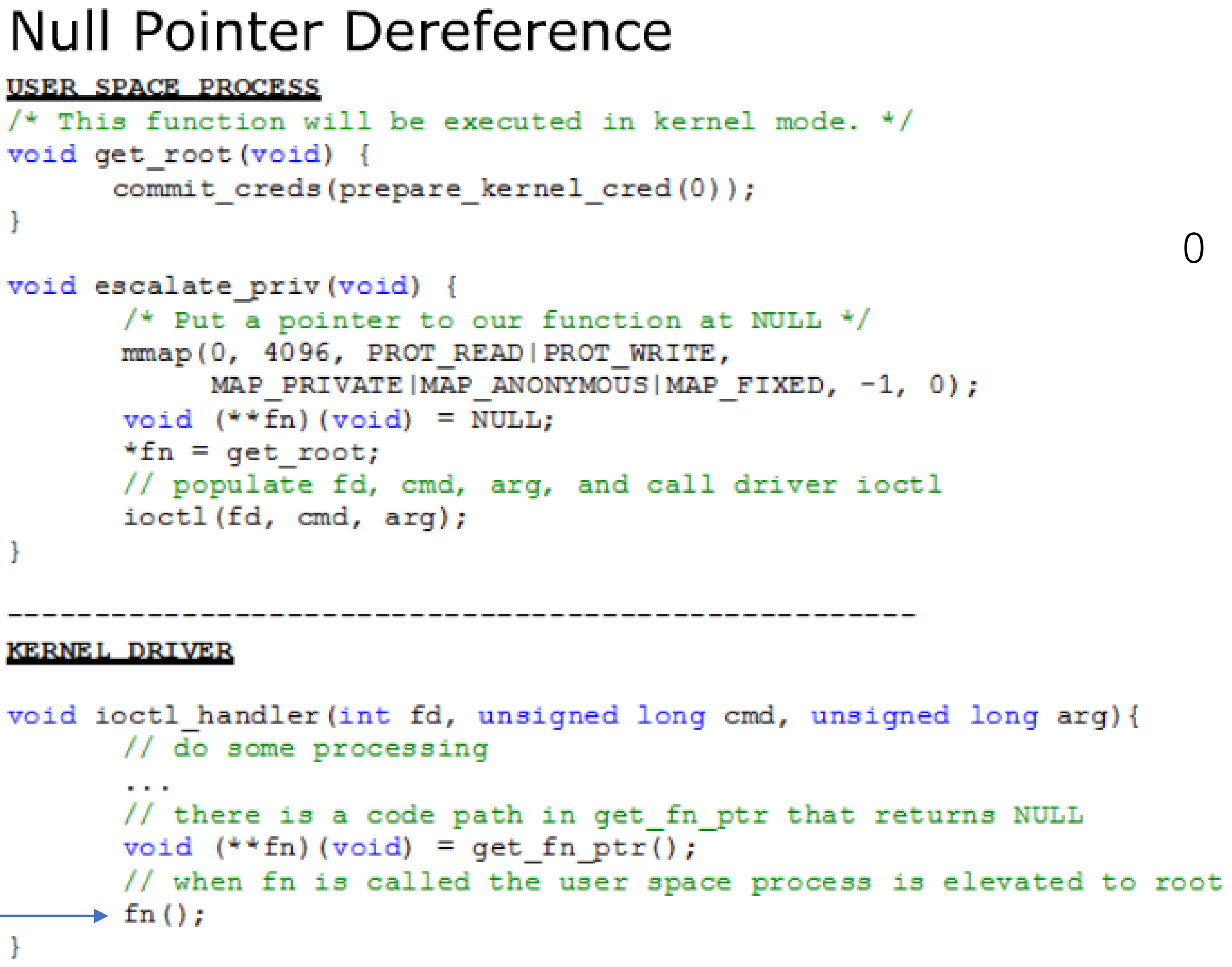
Input validations

- Verify that length fields match the amount of data provided
- Restrict text input to certain characters
- Restrict numeric data to acceptable range
- Verify conformance to standards

Problem with pointers

- Null pointers
  - Null pointer dereference of a function pointer in kernel.
  - If malicious user space process can map zero page, then it can put the address of a function such that when called by kernel module it will elevate the privilege of the process
  - Always check for NULL pointer

NULL pointers



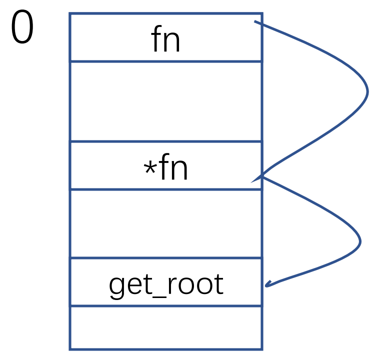
Null Pointer Dereference

USER SPACE PROCESS

```
/* This function will be executed in kernel mode. */
void get_root(void) {
        commit_creds(prepare_kernel_cred(0));
}

void escalate_priv(void) {
        /* Put a pointer to our function at NULL */
        mmap(0, 4096, PROT_READ|PROT_WRITE,
             MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
        void (**fn)(void) = NULL;
        *fn = get_root;
        // populate fd, cmd, arg, and call driver ioctl
        ioctl(fd, cmd, arg);
}

---------------------------------------------------
```

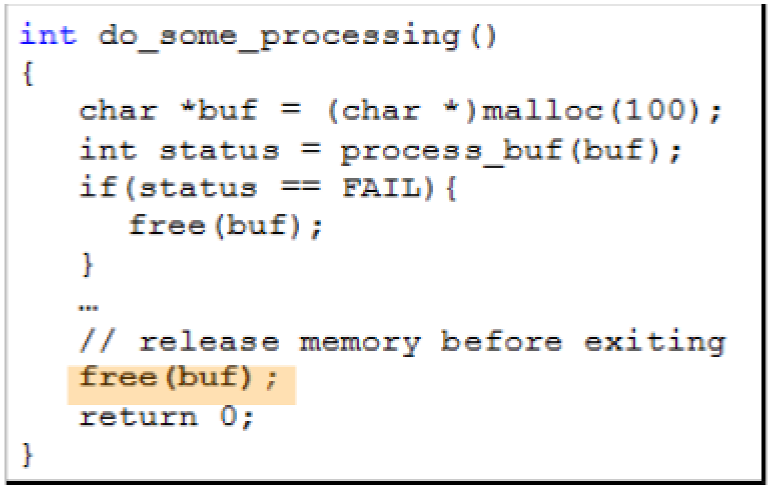
KERNEL DRIVER

```
void ioctl_handler(int fd, unsigned long cmd, unsigned long arg){
        // do some processing
        ...
        // there is a code path in get_fn_ptr that returns NULL
        void (**fn)(void) = get_fn_ptr();
        // when fn is called the user space process is elevated to root
        fn();
}
```



Double free

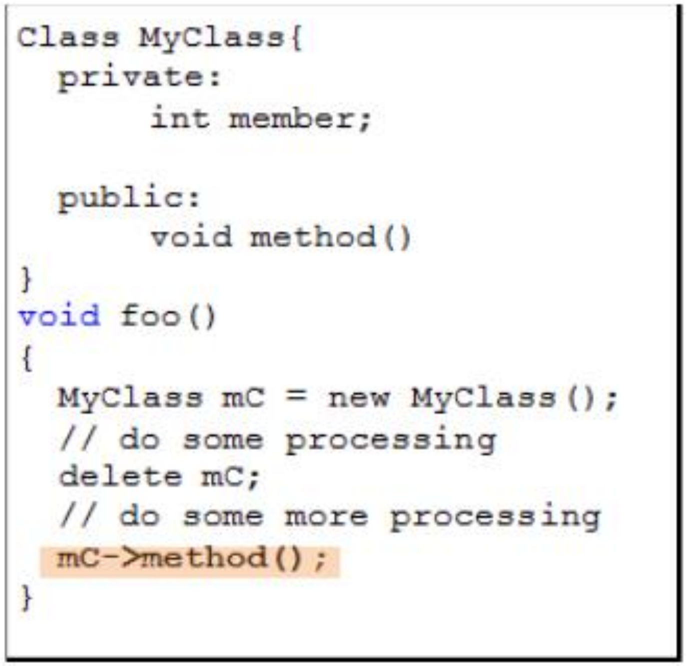
- Can cause heap corruption and lead to exploitation of heap management routines
- If attacker can place data of his choosing at the freed location, he can trick the program into executing malicious code.



```
int do_some_processing()
{
   char *buf = (char *)malloc(100);
   int status = process_buf(buf);
   if(status == FAIL){
      free(buf);
   }
   …
   // release memory before exiting
   free(buf);
   return 0;
}
```

Double free – pointer arithmatic

- In C++ the VTABLE of the malicious object could point to a location that contains functions controlled by attacker, so when freed pointer calls a function of the object, the program will not crash, but instead will call one of the functions from VTABLE.



```
Class MyClass{
  private:
        int member;

  public:
        void method()
}
void foo()
{
  MyClass mC = new MyClass();
  // do some processing
  delete mC;
  // do some more processing
  mC->method();
}
```

Sign conversion vulnerability

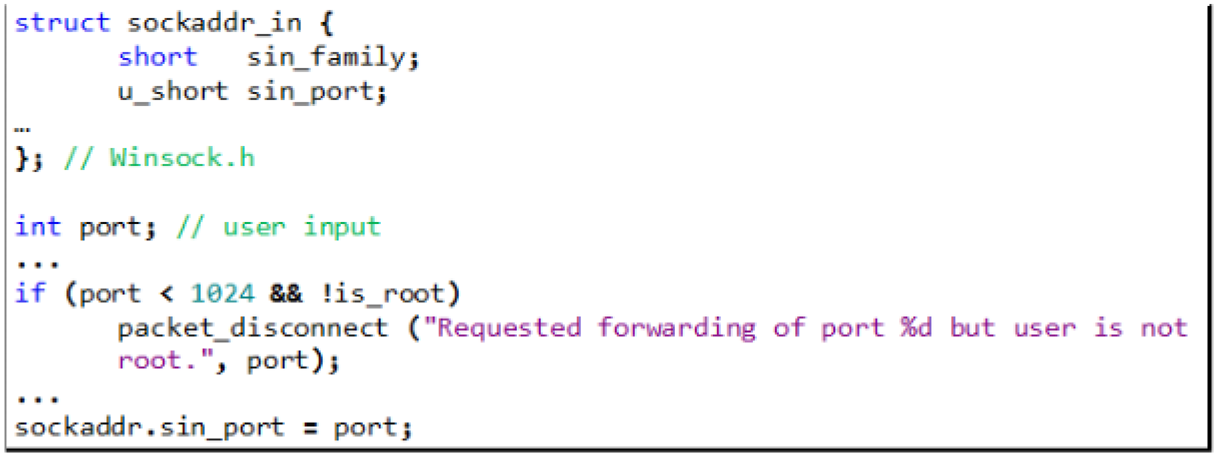
- Rules of type conversion are complex and non-intuitive. When data type of variable is changed, the sign or value of the variable may change unexpectedly leading to vulnerability.

```
  signed int i = -1;
unsigned int j = 1;
if (i < j) { printf("foobar"); }
```

  - Implicit type conversion by compiler forces i to be promoted to unsigned

If an attacker can induce the victim application to enter an *unintended state* using a type conversion, this is known as a type conversion vulnerability.

Sign conversion vulnerability



```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
…
}; // Winsock.h

int port; // user input
...
if (port < 1024 && !is_root)
        packet_disconnect ("Requested forwarding of port %d but user is not
        root.", port);
...
sockaddr.sin_port = port;
```

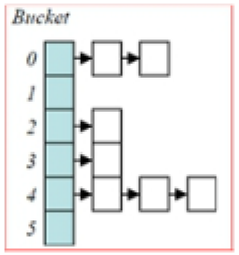
- The first 1024 ports are privileged ports and normal users cannot run servers on privileged ports (HTTPS, POP, SMTP). So if you connect to a service on one of these ports, the service is a legitimate service.
- In structure sockaddr_in, member sin_port is declared 16-bit unsigned integer (short).
- Port is declared 32 bit integer
- When port is assigned to sin_port, the two high-order bytes are truncated and port number is changed.
- To exploit this code, an attacker might set the variable port to 0x000101bb. The expression in if statement is satisfied, and sin_port is set to 0x1BB (443), which is used for HTTPS traffic.

Denial of Service

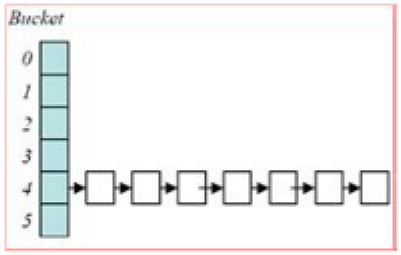
- Attacker forces you to deplete a resource, such as CPU cycles, memory, bandwidth, socket/port connections, battery, etc..
- Example:
  - In a normal case, a decoded message will be acted upon, but in an unexpected state, the message is dropped but not freed (bug)
  - An attacker could exploit this scenario by keep sending messages of the same type until ran out of memory

Denial of Service

- Hashes are convenient data structure that offer O(n) average time for n insertions, but could degenerate to O(n^2) time for n insertions.
- An attacker could control or predict the input used by hash algorithms to always induce the worse-case behavior



Normal



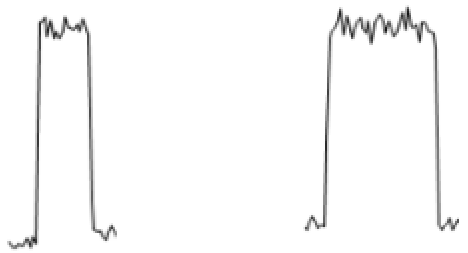
Worst-case

Denial of Service

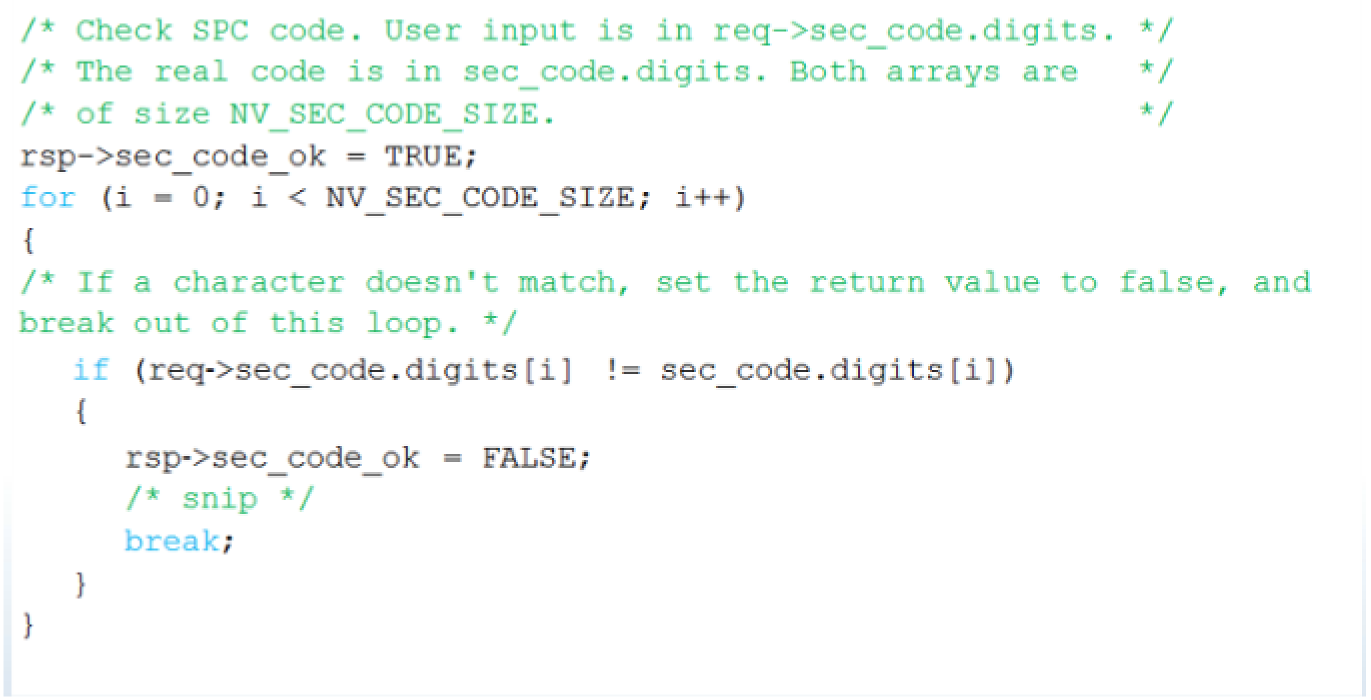
- DoS countermeasures
  - Be careful about resource consumption (file handles, memory, etc..)
  - Perform input validation of all user-controlled data. Prefer whitelist over blacklist
  - Fail gracefully with least impact.
  - Keep time complexity attacks in mind (average v.s worst-case)

Side channels

- Attacker can monitor the hardware for information that allows him to break into the system
  - Power, electromagnetic, timing
- Attacker could use a high precision timer to time how long it takes to run



Side channels



```
/* Check SPC code. User input is in req->sec_code.digits. */
/* The real code is in sec_code.digits. Both arrays are   */
/* of size NV_SEC_CODE_SIZE.                              */
rsp->sec_code_ok = TRUE;
for (i = 0; i < NV_SEC_CODE_SIZE; i++)
{
/* If a character doesn't match, set the return value to false, and
break out of this loop. */
   if (req->sec_code.digits[i] != sec_code.digits[i])
   {
      rsp->sec_code_ok = FALSE;
      /* snip */
      break;
   }
}
```

Mitigations

- Compartmentalization
  - Principle of least privilege
    - Process should have only the necessary privilege to perform their work and not more.
    - When privilege is no longer needed, it should be forfeited
  - Privilege separation
    - Memory virtualization
- Secure by default
- Keep it simple
- Minimize the system entry points
  - Open sockets, named pipes, device driver nodes, config files
- Fail securely
  - Log errors carefully, leave the system in a consistent state