The perfect hack!

-By IEEE NITK Systems and Security SIG

What do you think the perfect hack is?

Before we get started ..

1. Download the necessary files.
   a. git clone https://github.com/adwait1-G/ieee_workshop.git

2. Some program execution internals.

a. Download “peda” for gdb
b. How does a function call work under the hood..

3. Disable ASLR(a type of memory protection against attacks)

Then get started!

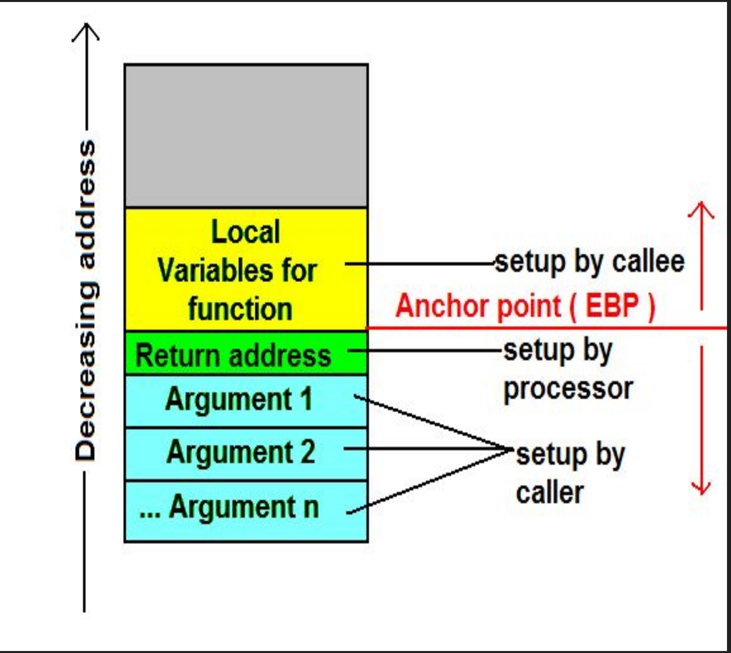
Program execution internals

1. $cd ieee_workshop/introduction
2. gcc func.c -o func -fno-stack-protector -m32
   a. This creates a 32-bit executable.
   b. This removes the stack protector

3. ./func

4. gdb -q func

So , this is how the stack looks like during function call.



Point to note:

1. The order in which everything is stored.
2. Lower Address , Local variables , Return Address , Arguments , Higher Address

Continued..

a. It might so happen that there are no local variables for a function.
b. So , the stack could have only
   i. Return Address
   ii. Arguments

VV IMPORTANT TO NOTE THIS!! .

Disabling ASLR:

```
$sudo echo 0 > /proc/sys/kernel/randomize_va_space
```

Objective :

“YOUR SERVER HAS BEEN HACKED.”

#whoami

root

Objective in real detail :

A linux machine is running a server.

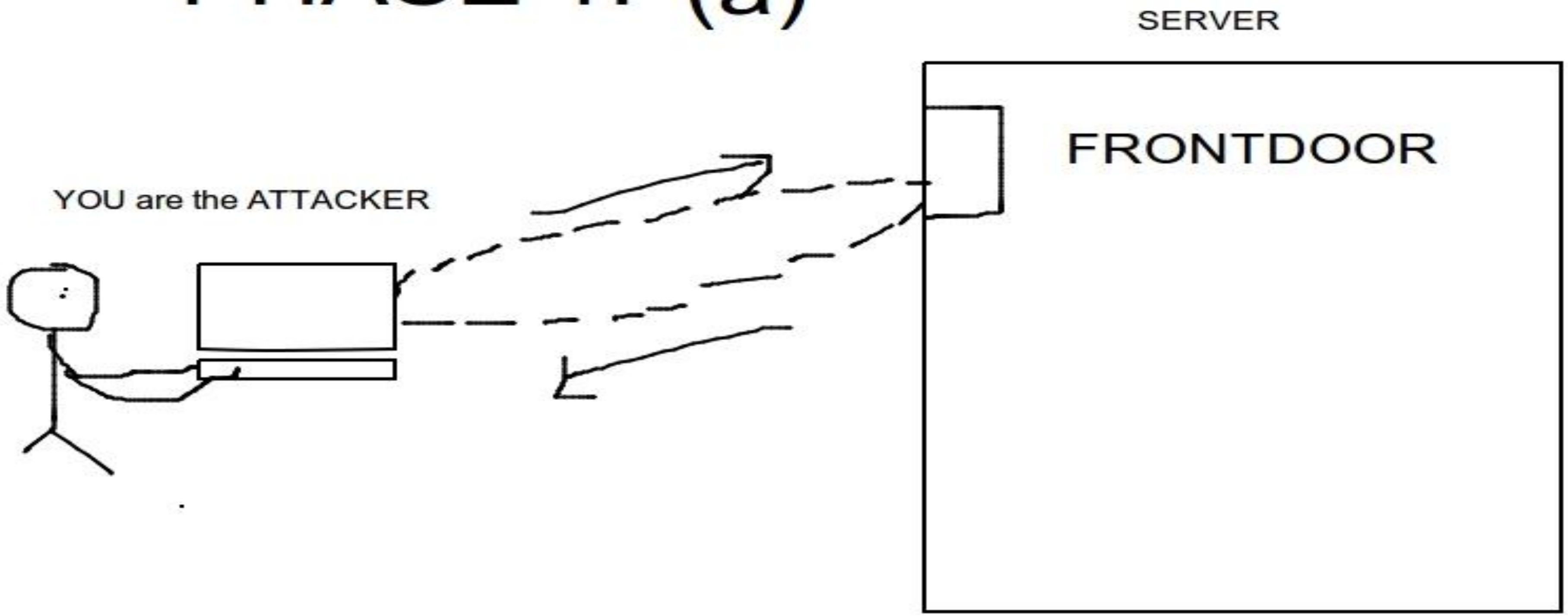
Phase 1:

1. Search around for bugs in the server program .
2. Get access to the system - create a “backdoor”

Phase 2:

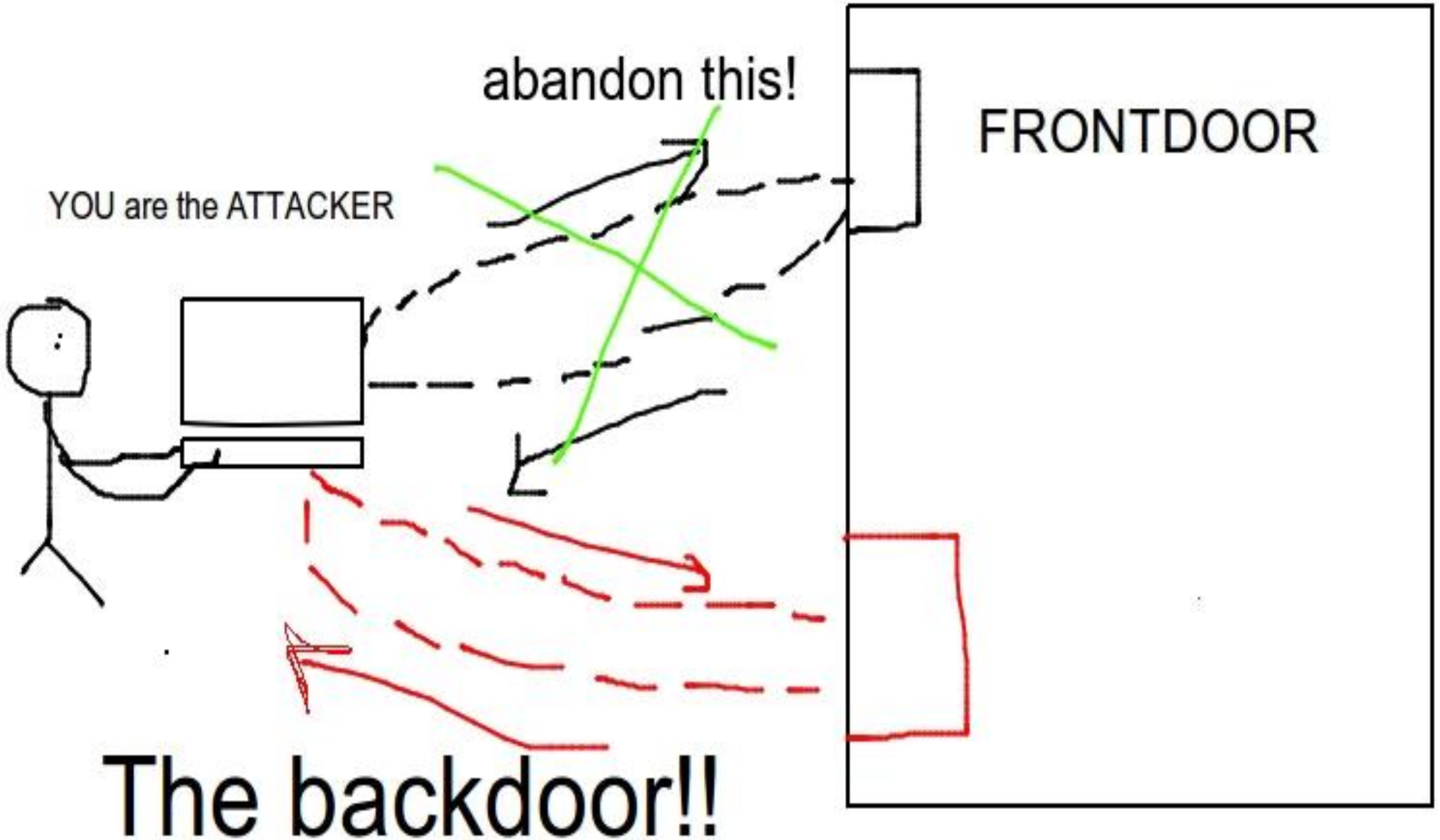
3. Look around some other buggy program.
4. System “pwned” - GET THE ROOT SHELL!

PHASE 1: (a)



ALL normal!

DO SOME MAGIC AND THEN...


abandon this!
FRONTDOOR
YOU are the ATTACKER
The backdoor!!

We will see what that magic is...

Phase 1: Creating backdoor

1. cd ieee_workshop/server_dir
2. Compile the server.c to get the server executable
   a. gcc server.c -o server -m32 -fno-stack-protector -zexecstack
3. Run the python script like this.
   a. python start_server.py
   b. python start_server.py 192.168.43.116 6000

The server is up and running in the terminal1 - T1.

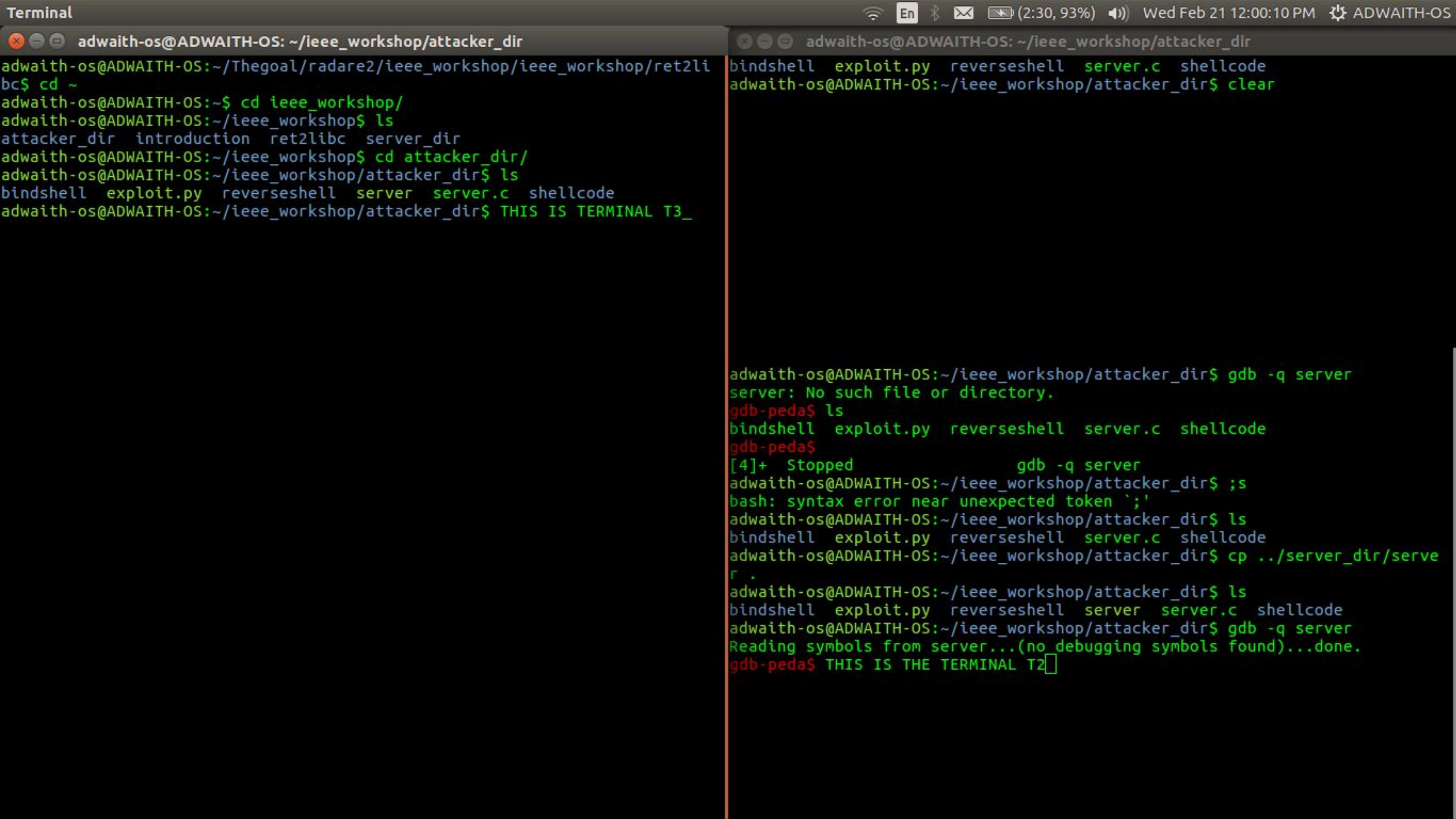
Open another terminal - Terminal2 - T2

1. cd ieee_workshop/attacker_dir
2. Copy that “server” executable and server.c sourcecode to the attacker_dir
   a. cp ../server_dir/server .
   b. cp ../server_dir/server.c .
   c. cp ../server_dir/start_server.py .
3. Start the “server” in debug mode.
   a. gdb -q server

4. We will debug the server application and find the vulnerability.

One more terminal , Terminal3 :

We will be working with Terminal2 and Terminal3 till we find the bug and exploit it.



```
Terminal                                   (2:30, 93%)  Wed Feb 21 12:00:10 PM  ADWAITH-OS
```

```
adwaith-os@ADWAITH-OS: ~/ieee_workshop/attacker_dir
adwaith-os@ADWAITH-OS:~/Thegoal/radare2/ieee_workshop/ieee_workshop/ret2li
bc$ cd ~
adwaith-os@ADWAITH-OS:~$ cd ieee_workshop/
adwaith-os@ADWAITH-OS:~/ieee_workshop$ ls
attacker_dir  introduction  ret2libc  server_dir
adwaith-os@ADWAITH-OS:~/ieee_workshop$ cd attacker_dir/
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ ls
bindshell  exploit.py  reverseshell  server  server.c  shellcode
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ THIS IS TERMINAL T3_
```

```
adwaith-os@ADWAITH-OS: ~/ieee_workshop/attacker_dir
bindshell  exploit.py  reverseshell  server.c  shellcode
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ clear







adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ gdb -q server
server: No such file or directory.
gdb-peda$ ls
bindshell  exploit.py  reverseshell  server.c  shellcode
gdb-peda$
[4]+  Stopped                 gdb -q server
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ ;s
bash: syntax error near unexpected token `;'
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ ls
bindshell  exploit.py  reverseshell  server.c  shellcode
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ cp ../server_dir/serve
r .
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ ls
bindshell  exploit.py  reverseshell  server  server.c  shellcode
adwaith-os@ADWAITH-OS:~/ieee_workshop/attacker_dir$ gdb -q server
Reading symbols from server...(no debugging symbols found)...done.
gdb-peda$ THIS IS THE TERMINAL T2
```

T2 - gdb terminal.

Follow my lead.

Why are we running 2 servers now???

1. The server in port 6000 depicts the original server.
2. The server with gdb at port 4000 depicts the server program we will debug and find the bug.

For now , leave the port 6000(original server) alone.

Note: Any 2 port numbers can be used .

T3 :

1. Let us examine the sourcecode.
2. And find where could we have a bug.
3. Check out the application() function part.

4. Functions like strcpy , memcpy , gets() are the root causes of such bugs.

What might happen if string is more than 250 bytes?

Let us go back to the gdb again and try.

Why are we getting the segfault??

1. The return address of the function might be getting overwritten by a part of our string.
2. Let us examine this .

Gdb again:

1. run 127.0.0.1 4005

The server segfaulted because the return address was overwritten by some other invalid address.

IDEA: What if we overwrite the return address with a valid address???

What we do is , load our code at the beginning of the buffer , and overwrite the return address with the starting address of buffer. Now , the function returns to the loaded code.

Shellcode...

1. The loaded code is known as shellcode.
2. The direct machine code is injected into the memory and is executed.

These type of vulnerabilities in a program are generally known as CODE INJECTION VULNERABILITIES.

Any doubts so far..

Creating a listening terminal.

We will be using what is known as reverse shellcode.

When this type of shellcode is executed , the server/victim will automatically start trying to connect with the attacker’s machine.

```
$ nc 192.168.43.116  -p 33333  -l -v
```

So , the exploited server can connect to this.

Fastforwarding...

We will use the exploit.py and finish it.

Phase 1 done.

Phase 2: “pwn” it

A. Only objective:

   GET THE ROOT SHELL!

Phase2 : Suppose you are inside the server.

1. $cd ieee_workshop/ret2libc

There is a binary file named “application”

a. $chmod u+x application
b. $chown root application
c. $chgrp root application
d. chmod u+s application

Phase2:

$ls -l application   .Do you see the “application” is colored with red?

Such applications are called setuid applications.

These are very dangerous applications. They are necessary evil.

Point to note:

1. When that application is run , that runs with root privileges.
2. If a shell is executed by that process , that shell will be a ROOTSHELL.

That is the idea!

Phase2:

1. Check for a bug.
2. In phase 1 , we used code injection .
3. Here , we will what is known as the return-to-library attack to get the rootshell.

Instead of overwriting return address with some stack address , we overwrite it with a C library function.

So that , that function is executed.

Is it that simple?

No.

1. A function should have a stackframe. Ie.,minimum it should have
   a. Return address
   b. Arguments(If present)

How do we create this FAKE stackframe??

Creating a fake stackframe:

Let the original stack of function look something like this:

Buffer + ReturnAddress + Arguments to that function

We will modify it as ,

Buffer + <address_of_a_library_function> + <Return address of library function> + <Arguments to the library function>

Creating fake stackframe:

Buffer + ReturnAddress + Arguments to that function

Buffer + <address_of_a_library_function> + <Return address of library function> + <Arguments to the library function>

4 bytes of ReturnAddress is replaced by Address_of_library_function

Next 4 bytes are replaced by Return_address_of_library_function

Next 4 bytes are replaced by 1_argument_to_library_function

Phase2:

1. The library function we chose is “system”
2. This will execute any shell command if given as an argument.

So , we need a shell. Let us do this:

system(“/bin/bash”)

Finding the addresses:

1. Using gdb.
2. gdb -q application
3. b main
4. run aaaaaa
5. print system :Peacefully prints the address of system() function.
6. print exit :prints address of exit() function.
7. Where can we find “/bin/bash” in the program??

Finding addresses :

1. x/500s $esp

Somewhere , SHELL=”/bin/bash” is present right?

But , to make the process easy for us , I have done this:

1. Run the application.
    a. ./application aaaaa

   It should print out the Address of “/bin/bash”

We have everything now.

The exploit: bruteforce.bash is a very simple script to exploit it.

Run it:

```
$./bruteforce.bash
```

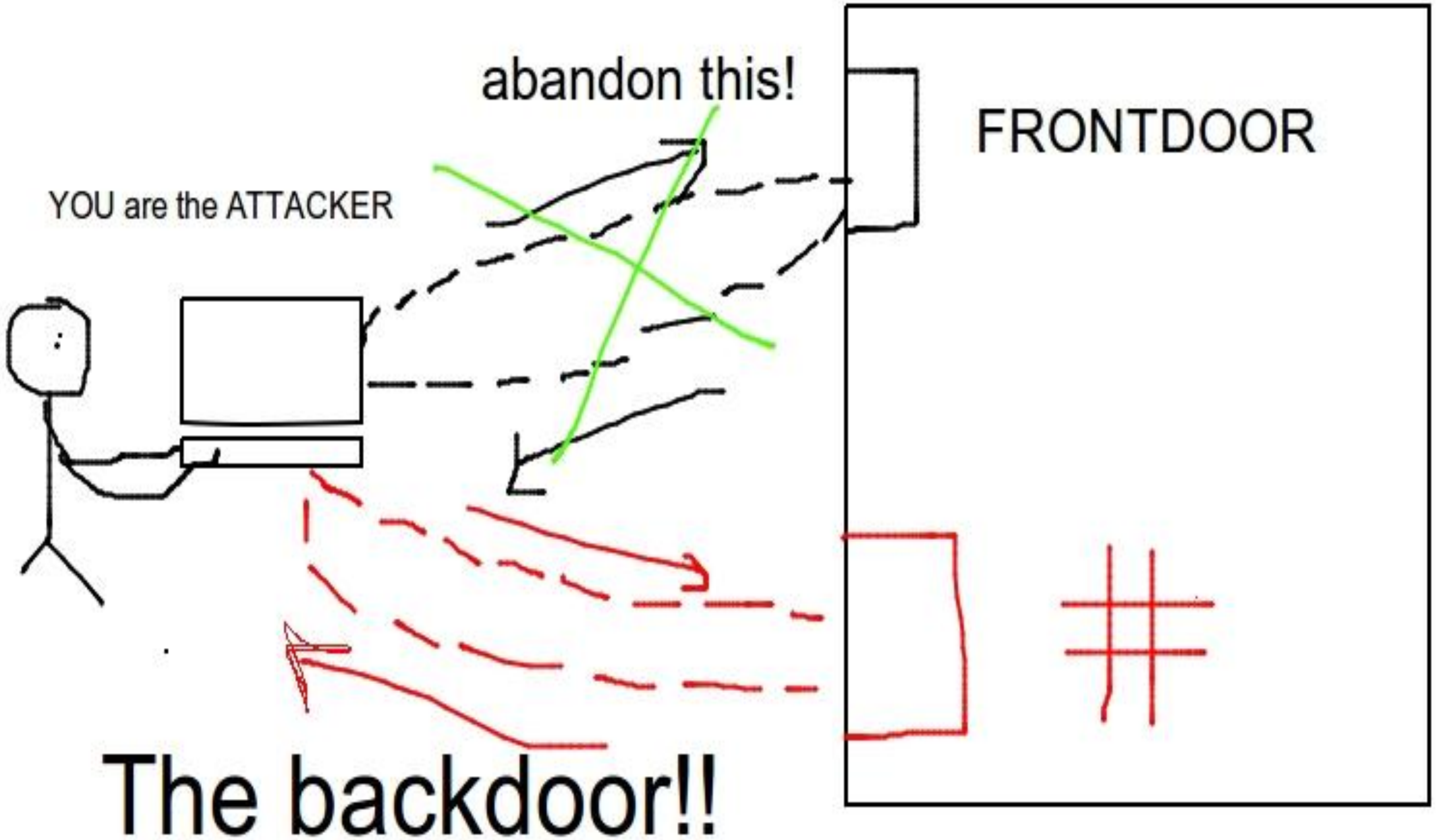
Make a note of the Address of “/bin/bash”

Make changes in the script and run it again.

Wait till you get the root shell..

Is it done??

And its done!


abandon this!
FRONTDOOR
YOU are the ATTACKER
The backdoor!!

Why do we need a root shell?

1. Getting absolute control of the machine.
2. Can install rootkits (the name says it , only root people can install it)
3. It’s hard but is fun getting the root shell :P

These are a few real-life vulnerabilities :

1. https://nvd.nist.gov/vuln/detail/CVE-2017-6009
2. https://www.cvedetails.com/cve/CVE-2017-1000176/
3. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9678
4.

And many more...

Some more stuff to show:(if time permits)

1. What code did we run to create the backdoor??
2. Types of backdoor
3. Writing shellcode

Any questions so far?

Any feedback , suggestions etc., ?

Thank you !
Keep hacking!