Project 2: Return-oriented Programming

Due: Friday, November 3, 2017, 23:59

Goal

The goal of this project is to practice writing return-oriented programs and implement several different types of shellcode.

Specifically, there is one `target` binary which is a statically-linked, 32-bit x86 Linux binary. You will write five different Python programs which will exploit the binary in different ways. Unlike Project 1, this target is a server program which you will treat as if it were running on a remote machine. (In reality, both the `target` and the exploits will run on the same machine, but they will communicate over TCP.)

Skeleton code for each Python program has been provided, but each will need substantial modification.

Collaboration

You may work on this project in collaboration with a single partner as described on the main page.

You must not discuss the project with anyone other than your partner and course staff. You may use online resources for general reference, but not to search for solutions to specific questions posed in this project.

The Environment

You should be able to work in any 32- or 64-bit Linux environment that has Python 2.7 (and netcat is helpful). In particular, you may use `bertvm` or `ernievm`.

When exploiting the target (see below), you should not assume anything about stack addresses (in particular, you should not assume anything about environment variables).

The Target

The `target` binary has been compiled from `target.c` and statically linked with PCRE2. You must not change `target` or attempt to recompile it from `target.c`. If you do so, your exploits will almost certainly not work against the unmodified `target`.

The `target` is a simple server program that listens on the specified port for a connection. Once connected, a client can send one of two commands:

```
PUT SECRET <password> <secret>\r\n
```

and

```
GET SECRET <password>\r\n
```

The first command directs the server to remember a secret and its associated password. (The password may not contain white space.) The second command directs the server to reply with the secret, if the password matches. See `target.c` for the precise workings.

The Assignment

There are five tasks to be completed. Each task involves modifying a Python exploit program to construct a return-oriented exploit to exploit `target` in the manner specified.

WARNING: Although care has been taken to limit connections to localhost, as part of this project, you will be connecting sockets to shells. You *must* ensure that all sockets you create, either in the Python exploit programs or in `target`

are bound to `127.0.0.1`.

FURTHER WARNING: Any attempt to connect to other students' open ports is strictly prohibited. Violations will result in a 0 for the project and academic integrity hearings. So please, just don't do it.

Unlike Project 1, code injection will not be possible due to standard DEP protections. Instead, you will need to implement return-oriented programs. This will be tricky, but it is how modern software exploitation actually works.

To find gadgets in `target`, you may use Jonathan Salwan's `ROPgadget` tool. `ROPgadget` requires the `capstone` disassembler to run. This is not installed on the departmental servers, but you should be able to install it on your own machines. We have included a `gadgets.txt` file that is the result of running:

```
$ ./ROPgadget.py --binary target --ropchain > gadgets.txt
```

You may still find it helpful to run it yourself with other options.

Each task description below contains an example which consists of running `./target <port>` in one shell (with `<port>` replaced by a real port number) and running the Python program for the task. In addition, each task description includes the partial output of running the target using strace(1). Your code should induce the same behavior in `target`. That is, replacing `./target <port>` with `./strace target <port>` and following the provided example should produce the same system calls, with the same arguments, in the same order as shown. (File descriptor numbers may be different from those shown, but should be self-consistent.)

You may not use any of the system calls that allocate writable and executable memory nor any system calls that change memory page permissions. In particular, all of the exploits you write must be entirely in the return-oriented style; you should not inject any x86 code.

The Tasks

Task 1 Local shell (`local.py`)

The first task is to exploit `target` and have it exec a shell. Conveniently, `ROPgadget` has constructed such an exploit for us! Look at the bottom of `gadgets.txt` (or run `ROPgadget.py` with the `--ropchain` option).

The constructed exploit uses `struct.pack` to build a binary string. See the documentation for details. You're going to be using this function extensively.

Let's take a look at the first few calls.

```
p += pack('<I', 0x0808522a) # pop edx ; ret
p += pack('<I', 0x08139060) # @ .data
p += pack('<I', 0x080f1016) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080c219d) # mov dword ptr [edx], eax ; ret
```

Each 4-byte word in the `p` string is either the address of code to return to or some data. The first word, `0x0808522a` is the address of the instructions `pop edx ; ret`, as noted in the comment. When `target` returns to this address, it will pop `0x08139060` into `edx`.

But what is that value and what does `@ .data` mean? If you run `readelf -S target`, you'll see that the `.data` section starts at address `0x08139060`. `ROPgadget`'s shellcode has decided to use `target`'s writable data section as a place to write some data. In particular, this will write `/bin` at `0x08139060`. You'll want to keep this in mind for some of the other parts.

At this point, you should try to figure out what the rest of the code is doing. You can probably get away with not understanding this, but it'll make the whole rest of the project easier if you figure this out now.

One final point about the generated shellcode: The final 13 words clear `eax`,

increment `eax` 11 times, and then run `int 0x80`. `ROPgadget` is avoiding zero-bytes. Since you are using sockets which can handle binary data, this isn't a concern. Go ahead and replace the `xor` and the `inc`s with a `pop eax`.

To complete this task, modify `local.py` to:

1. Exploit `target` and make it exec `/bin/sh` by overwriting the saved `eip` and the subsequent words on the stack with this return-oriented program. (Disassembling `target` using `objdump -d target` can help you figure out where the saved `eip` is relative to the start of the array.)

To test that everything works, run `./target <port>` in one shell and in another shell, run `./local.py <port>` (replace `<port>` which an actual port number). You should see something like the following.

```
user@bertvm:~/project2$ ./target <port>
$
```

Running `strace ./target <port>`, with the example above prints out (in part)

```
user@bertvm:~/project2$ strace ./target <port>
...
send(4, "INVALID COMMAND\r\n", 17, 0)   = 17
execve("/bin//sh", [], [/* 0 vars */])  = 0
...
```

Task 2 Dup shell (`dup.py`)

The first exploit was fun to do (I hope), but not terribly useful. After all, it opened a shell on the "remote" machine with no way to communicate with it! You're going to fix that right now.

You need to connect `target`'s `stdin`, `stdout`, and `stderr` to the socket before you exec `/bin/sh`. Fortunately, that's easy to do with the dup2(2) system call.

To make a system call, you need to know what to put in each register. Fortunately, kernelgrok is a great resource. Search for `dup` to see what goes in each register.

One tricky aspect is you need to put the socket file descriptor in register `ebx`, but you can't know what value to use until the exploit connects. Looking at the disassembly for `target`, you'll see that the return value from accept(2) is stored in `ebx` in `main` and also on the stack just above the saved `eip` as the first argument to `handle_connection`. Unfortunately, you're going to trash the saved `ebx` as well as the argument. All hope is not lost, run `target` in `gdb` and break near the end of `handle_connection`. Luckily, the socket file descriptor is available!

In essence, you want to get the socket file descriptor that was returned from accept(2) in `target`—call it `sock`—and make the three system calls that correspond to

```
dup2(sock, 0);
dup2(sock, 1);
dup2(sock, 2);
```

and then exec `/bin/sh` as you did in `local.py`.

To complete this task, modify `dup.py` to:

1. Exploit `target` and have it perform the dup2(2) system calls and exec the shell as described above;
2. Read from `stdin` and write to the socket and read from the socket and write to `stdout`. You may find the `console` function in `console.py` useful for this task. Simply import the function using `from console import console`, pass the socket to `console`, and it should take care of everything. The prompt will not appear, but you can still enter commands and see the result.

To test that everything works, run `./target <port>` in one shell and in another shell, run

```
user@bertvm:~/project2$ ./dup.py <port>.
INVALID COMMAND
date
Sun Oct 16 03:28:26 CDT 2016
exit
```

Running `strace ./target` as described in the hints, with the example above prints out (in part)

```
user@bertvm:~/project2$ strace ./target <port>
...
send(4, "INVALID COMMAND\r\n", 17, 0)   = 17
dup2(4, 0)                              = 0
dup2(4, 1)                              = 1
dup2(4, 2)                              = 2
execve("/bin//sh", [], [/* 3 vars */])  = 0
...
read(0, "date\n", 8192)                 = 5
...
read(0, "exit\n", 8192)                 = 5
...
```

Task 3 Reverse shell (`reverse.py`)

The exploit used in `dup.py` connected the shell to the socket we used to connect to `target` initially. For this task, the exploit will cause `target` to make a connection to remote server, connect the resultant socket to `stdin` / `stdout` / `stderr` (as was done in `dup.py`), and exec a shell.

Creating a new socket and making a connection involves making several system calls.

- socket(2)
- connect(2)

There are several ways to call these functions. Since they appear in `target`, it's

possible to return to them with the arguments on the stack. However, the first argument to connect(2) is the return value from the socket(2) which makes making returning to the libc implementations difficult. Instead, you should make the corresponding system calls directly; just as you did with dup2(2).

See the associated manual pages for example usage and see the hints below for suggestions on making these system calls. And see the example below for the arguments to the system calls.

The connection should be to `127.0.0.1` and the port should be specified as an argument to `reverse.py` (see example below).

To complete this task, modify `reverse.py` to:

1. Open a socket to listen on `127.0.0.1` with the port specified as a command line parameter to `reverse.py` (see example below).
2. Exploit `target` and have it make a new connection to `127.0.0.1` with the same port used in step 1. Once connected, `target` should exec a shell with `stdin` / `stdout` / `stderr` connected to the new socket.
3. Read from `stdin` and write to the newly opened socket and read from the socket and write to `stdout`. Again, the `console` function may be helpful.

To test that everything works, run `./target <port>` in one shell and in another shell, run

```
user@bertvm:~/project2$ ./reverse.py <port> <connect_port>.
INVALID COMMAND
date
Sun Oct 16 03:47:33 CDT 2016
exit
```

Running `strace ./target <port>`, with the example above prints out (in part)

```
user@bertvm:~/project2$ strace ./target <port>
...
```

```
send(4, "INVALID COMMAND\r\n", 17, 0)    = 17
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 5
connect(5, {sa_family=AF_INET, sin_port=htons(<connect_port>), sin_
dup2(5, 0)                              = 0
dup2(5, 1)                              = 1
dup2(5, 2)                              = 2
execve("/bin//sh", [], [/* 6 vars */])  = 0
...
read(0, "date\n", 8192)                 = 5
...
read(0, "exit\n", 8192)                 = 5
...
```

Task 4 Bind shell (`bind.py`)

The exploit used in `dup.py` connected the shell to the socket we used to connect to `target` initially; the exploit used in `reverse.py` made a connection back to the attacker. For this task, the exploit will cause `target` to start listening on a specified port and when a connection happens, connect `stdin` / `stdout` / `stderr` to the newly connected socket and exec a shell. This is called a *bind shell*.

Listening on a port and accepting new connections involves making several system calls.

- socket(2)
- bind(2)
- listen(2)
- accept(2)

As was the case for `reverse.py`, there are several ways to call these functions. Since they're used in `target`, it's possible to return to them with the arguments on the stack. However, the first argument to each of the last three require system calls requires the return value from the first which makes making returning to the libc implementations more difficult. Instead, you should make the corresponding

system calls directly.

See `target.c` and the associated manual pages for example usage and see the hints below for suggestions on making these system calls. And see the example below for the arguments to the system calls.

Note that the bind(2) call *must* bind the socket to address `127.0.0.1` as mentioned in the warning above.

To complete this task, modify `bind.py` to:

1. Exploit `target` and have it listen for a new connection on a different port (specified as a command line parameter to `bind.py`—see example below). Once a connection happens, `target` should exec a shell with `stdin` / `stdout` / `stderr` connected to the socket.
2. Make a new connection to `127.0.0.1:<listen_port>` where `<listen_port>` is the port specified as the second command line parameter (see example).
3. Read from `stdin` and write to the newly opened socket and read from the socket and write to `stdout`. Again, the `console` function may be helpful.

To test that everything works, run `./target <port>` in one shell and in another shell, run

```
user@bertvm:~/project2$ ./bind.py <port> <listen_port>.
INVALID COMMAND
date
Sun Oct 16 03:47:33 CDT 2016
exit
```

Running `strace ./target <port>`, with the example above prints out (in part)

```
user@bertvm:~/project2$ strace ./target <port>
...
send(4, "INVALID COMMAND\r\n", 17, 0)   = 17
```

```
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 5
bind(5, {sa_family=AF_INET, sin_port=htons(<listen_port>), sin_addr=
listen(5, 135499884)                    = 0
accept(5, 0, NULL)                      = 6
dup2(6, 0)                              = 0
dup2(6, 1)                              = 1
dup2(6, 2)                              = 2
execve("/bin//sh", [], [/* 3 vars */])  = 0
...
read(0, "date\n", 8192)                 = 5
...
read(0, "exit\n", 8192)                 = 5
...
```

Task 5 Data exfiltration (`secret.py`)

The final task is to convince `target` to write its secret to the socket without knowing the password. No need to get a shell this time.

It's possible to write to a socket using send(2) or write(2). As such, there are several possible approaches that one could take: (1) return to the `__libc_send` function, (2) return to the `__libc_write` function, (3) make a send(2) system call, or (4) make a write(2) system call. Each approach requires passing the socket file descriptor. Since the file descriptor is not constant, returning to the `__libc_send` or `__libc_write` functions is actually more difficult than making a system call. As you've seen, making socket system calls is more complicated than other system calls, so go with approach (4).

You also need to know how much data to write. To do this, you're going to need to implement a return-oriented loop. This is easily the most complicated part of writing a return-oriented program. There are many ways that one could implement it. The method I used was to implement the following `StringLength` gadget:

1. Set `edi` to `0xffffffff`.

2. Set `esi` to point to a `0` byte. (I set `esi` to point to `.data` and wrote a `0` there.)
3. Set `ecx` to point to the start of the `secret` array.
4. Set `edx` to the number of bytes to conditionally move the stack pointer to start the next iteration of the loop (this should be a negative number).
5. Use `edi` as the length counter and `ecx` as a pointer to the next byte to test.
6. Use `eax` and `ebp` as scratch registers to perform the conditional update of `esp`.
7. After a short setup, the body of the loop increments the `ecx` pointer and `edi` length, compares the byte pointed to by `ecx` to 0, and if it is nonzero, branch back to the top of the loop by updating `esp`.
8. After the loop exits, `edi` holds the length of the string.

The sequences of code I used to implement step 6 of the `StringLength` gadget (organized by functionality, not order in which I used them) were

```
# Increment/decrement pointer and length.
0x080545a1 # dec ecx ; ret
0x0807703f # inc ecx ; mov ebp, 0xa0b80810 ; ret
0x0808d7c5 # inc edi ; ret

# Compare to 0.
0x08048ac1 # xor eax, eax ; ret
0x08127ded # cmp byte ptr [ecx], al ; ret

# Conditional update of esp.
0x0808522a # pop edx ; ret
0x0809c909 # cmovne eax, edx ; ret
0x080485f4 # pop ebp ; ret
0x0807c05f # add ebp, eax ; retf
0x0812c918 # add esp, ebp ; add cl, byte ptr [esi] ; adc al, 0x43 ;
```

There a few things to notice about these code sequences.

- `cmovene eax, edx` is a conditional move from `edx` to `eax`. See the Intel

manual for details.

- `retf` is a far return. The Intel manual has the complete details, but in short, it will pop two words off the stack. The first will go into `eip` and the second into the `cs` segment register. You should use `0x00000023` as the value for `cs`. In other words, if you want to use `add ebp, eax ; retf` followed by, for example, `pop edx ; ret`, you should use

```
p += pack('<I', 0x0807c05f) # add ebp, eax ; retf
p += pack('<I', 0x0808522a) # pop edx ; ret
p += pack('<I', 0x00000023) # cs
p += pack('<I', 0xdeadbeef) # edx = 0xdeadbeef
```

- `add esp, ebp ; add cl, byte ptr [esi] ; adc al, 0x43 ; ret` modifies `ecx` unless you set `esi` to point to a `0` byte as mentioned in step 2, above.
- None of those code sequences use `ebx`. Conveniently, `ebx` is the file descriptor argument for write(2). So if you first move the socket file descriptor into `ebx`, you can compute the length without disturbing it.

To complete this task, modify `secret.py` to:

1. Exploit `target` and have it write the secret value to the socket by making a write(2) system call. Afterward, exit cleanly by making an `exit` system call with return value 0. (See the hints below for suggestions on making multiple system calls.)
2. Read the secret from the socket and print it out. (The secret won't have a newline, so you'll probably want to add that yourself.)

You may assume that a secret has been set.

To test that everything works, run `./target <port>` in one shell and in another shell, run

```
user@bertvm:~/project2$ echo 'PUT SECRET password1 Secret value!' |
```

```
SECRET STORED
user@bertvm:~/project2$ ./secret.py <port>.
INVALID COMMAND
Secret value!
```

Running `strace ./target <port>`, with the example above prints out (in part)

```
user@bertvm:~/project2$ strace ./target <port>
...
accept(3, 0, NULL)                      = 4
recv(4, "PUT SECRET password1 Secret valu"..., 1024, 0) = 36
send(4, "SECRET STORED\r\n", 15, 0)      = 15
close(4)                                = 0
accept(3, 0, NULL)                      = 4
...
send(4, "INVALID COMMAND\r\n", 17, 0)    = 17
write(4, "Secret value!", 13)           = 13
_exit(0)                                = ?
```

Hints

- To disassemble `target`, use `objdump -d target`.
- `ROPgadget` is handy, but a little limited. In particular, starting with the second task, you'll need to make multiple system calls. The `int 0x80` gadget `ROPgadget` finds probably won't work for you.
- strace(1) is an invaluable tool. Use it to run `target` to see what system call arguments you're passing.
- The socket-related system calls all use the same system call number `sys_socketcall`. As described on kernelgrok, `eax` is set to `0x66`. `ebx` is set to the actual socket call you want to make. See the kernel source for the possible values. Lastly, `ecx` points to an array of the arguments to pass.
- Multiple calls to send(2) can be received by a single call to recv(2). If that's not what you want, put a small delay between send calls.
- Start early!

Deliverables

- You must include a file called `ID` which contains the names of all partners (or just your own if you worked by yourself).
- You will submit using the provided `submit.sh` script.