

Our Process for owning the Challenges in Applied Security's Hack IT 2.0 at Shmoocon 2008

Adam Pridgen and Ryan Smith
[adam.pridgen, ryanwsmith] -at- gmail.com
2/27/2008

This paper discusses our team's process and methodology for tackling each of the challenges except challenge 2 and 7. We discuss the tools we used and how we ran them along with a discussion about how to perform Basic Differential Analysis.

Introduction

These posts are almost 2 weeks out of date, but I still wanted to post them anyway. This post is just a collection of thoughts about how we went about solving the challenges. We took second place, and we got a nice new shiny iPhone (and yes it is now hacked). The paper is laid out in the following manner. The first two sections talk about Challenge 1, where the first section is devoted to how we went about solving the Challenge, and the second section is dedicated to basic differential analysis. The remainder of the paper is laid out with each challenge and how we went about solving them along with the tools we used.

Challenge 1: Help Bruce Fix His Grades

In this challenge we needed to telnet in and provide a string an input string that would set all of Bruce's grade to be OA. After 32 characters were entered, the system would either return a modified set of grades if all the inputs were valid hexadecimal characters, or it returned an error message. To see my solution methodology, see [Basic Differential Analysis](#). Basically, once we figured out how to input strings into the system, we used ipython, which is a Python console program, and wrote a quick Python script to rapidly test and observe each string set. The Python script we used for testing is shown in Table 1 below.

Table 1: Python Code used in conjunction with iPython in Challenge 1

```
import socket

# Initialize Grades
biology = '00000000'
history = '00000000'
english = '00000000'
physed = '00000000'

# initialize the connection
s = socket.socket()
s.connect(('victim.hackit',10001))
s.recv(1024)

def send_grades():
    s.send(biology+history+english+physed)
    print s.recv(1024)

# Now play with the values until you get an OA in all the fields
# set values like so:
biology = 05050505
# submit values like so:
send_grades()
#tada

# biology = 05050505
# history = 28282828
# english = Don't remember
# physed = Don't remember
```

In iPython, we would manually set string values based on the inputs we got back from the server. For example, we would set the value `biology = 'FFFFFFFF'` then look at the output we got back from the server corresponding to the Biology grades, which was `'F0F0F0F0.'` Next, we set the value of `biology = '00000000'` and we got an output of `'0F0F0F0F.'` Score. From this pattern, we knew that the function to create the Biology grades was $H(X) = X \oplus 0F0F0F0F$, so we set the final value of `biology = '05050505'` and got the output `'0A0A0A0A.'` Then we moved on to testing the rest of the values. We talk more about the basic differential analysis process in the next section, Basic Differential Analysis.

Basic Differential String Analysis

In support of my “HowTo” on the HackIT 2.0 at Shmocon, I thought I would put together a basic tutorial on performing basic differential cryptanalysis on very simple ciphers/obfuscation algorithms. This tutorial assumes you are knowledgeable in bit level logic, such as shifting left and right, XOR, AND, and OR operations, but we will do a quick review. The tutorial will focus on the byte level not the bit level because it makes everything much easier to read (staring at bits gets tedious ;)). In the tutorial, we will focus mainly on identifying and creating high level functions based on the output of another function, which in turn helps us craft an output string to meet our needs. Again, this is just a basic introduction.

First let's look at identifying a basic function by first examining the core operations bit operation NOT, XOR, OR, and AND.

NOT Operations

A NOT (!) operation is simply a bit flip where we have : $NOT(X) = !X$, $!1 = 0$ and $!0 = 1$. Identifying this can be fairly straight forward. Basically, if the input is the same and the output is the exact opposite of what was inputted, we might be able to infer a NOT operation on the value.

For example, if $X = x0F$ with $F(X) = xF0$ and $X = xF0$ with $F(X) = x0F$, this could be indicative of a NOT operation on the input.

XOR Operations

An XOR (^) operation is defined by the following: $1 \wedge 1 = 0$, $1 \wedge 0 = 1$, and $0 \wedge 0 = 0$. Identifying this can be fairly straight forward. Basically, if the input is the same as the XOR'ed value or the inputs are zero, then the output is 0. For example:

Let $X = x0F$ and $H(x0F) = x00$, then $F = x0F \wedge x0F = x00$, and if the input is $x00$, then the result would be $0F$ (e.g. $F(x00) = x0F$. $x00 \wedge x0F = x00$).

OR Operations

An OR ($|$) operation is defined by the following: $0|1 = 1$, $0|0 = 0$, $1|1 = 1$. Thus, if any input is OR'ed to a value in the function, if any of the bits in either value the result is one, otherwise it is zero. Here are some examples:

$X = x0F$ and $H = X | xF0$, $Y = xFF$, and if $X = xF0$ and $H = X | xFF$, then $Y = xFF$, and so on.

AND Operations

An AND ($\&$) operation is defined by the following: $0\&1 = 0$, $0\&0 = 0$, $1\&1 = 1$. Thus, if any input is AND'ed to a value in the function, if either bit is 0 then the result is 0. Here are some examples:

$X = x0F$ and $H(X) = X \& x00$, $Y = x00$, and if $X = xF0$ and $H(X) = X \& xFF$, then $Y = xF0$, and so on.

A Basic Example

Now as stated above, we want to identify a secret string, but the function H employs some function on our inputs to get Y . Let us play with some values and try to identify the function. For this demonstration, assume the values in the function are either 0 or F.

Let $X = x0000$ and $H(x0000) = x0F0F$

Based on the outputs and our knowledge from above, we can quickly realize that the following operations could be applied to the fields in the following manner: $(\&^|)(|^)(\&|^)(|^)*$. As you can see, some of the fields can have more than one operation applied to them. Note, a byte field here is represented with $()$ and operations that could be used to obtain the values in the fields are contained in the $()$. For example in the first field an AND, OR, or XOR operation could be used to obtain the value 0. From above, $0^0 = 0$, $0|0 = 0$, and $0\&0 = 0$, so more testing is warranted. Since this is a basic introduction, we treat each byte independently (e.g. each byte field corresponds to the respecting byte field). Continuing on, let's try another input:

Let $X = xFFFF$ and $H(xFFFF) = xFOFF$

Now we have narrowed down the possible operations for each field to $(|^)(\&|^)(|^)$. There is still no definite answer for fields 1 or 3, but these cases can be considered identities, meaning the input is always equal to the output (e.g. $1|1 = 1$ or $1^1 = 1$). Additionally, there is no need to identify the exact operation, since we are only trying to acquire a *global deduction*, which is the high level function that gives the same output as the original function for each input.

Another Example

Now let's do a more difficult example. Suppose we want to solve the following function where $H(X) = DEADB33F$, but we don't know the function, and we don't want to spend time brute forcing the values (we don't really learn that way ;)).

Step 0: The initial value: $H(xB4ADF00D) = x74FCFA03$

Step 1: Enter all Zeros: $H(x00000000) = xC0508A0E$

So here is what we learned:

Field 0: $0 (| \wedge) Z = C \Rightarrow$ Value must XOR or OR value to get something from 0, thus $? = C$

Field 1: $0 (& | \wedge) ? = 0 \Rightarrow$ If the result is 0, we need more tests

Field 2: $0 (| \wedge) ? = 5 \Rightarrow ? = 5$

Field 3: $0 (& | \wedge) ? = 0$

Field 4: $0 (| \wedge) ? = 8 \Rightarrow ? = 8$

Field 5: $0 (| \wedge) ? = A \Rightarrow ? = A$

Field 6: $0 (& | \wedge) ? = 0$

Field 7: $0 (| \wedge) ? = E \Rightarrow ? = E$

Step 2: Enter all F's: $H(FFFFFFFF) = 3FAE F5B1$

So we know $(\wedge) (& |) (\wedge) (|) (|) (\wedge) (&) (\wedge)$, and we can induct the following results:

- Field 0: $F (\wedge) C = 3 \Rightarrow$ Operation is XOR, because OR would result in an F
- Field 1: $F (& |) ? = F \Rightarrow$ Identify Value (input == output), $? = \text{input}$
- Field 2: $F (\wedge) 5 = A \Rightarrow$ Operation = XOR
- Field 3: $F (&) ? = D \Rightarrow$ Operation = AND, because $(F | 0) == (F \wedge 0) == F$. Additionally, $? = D$ because $(F \& D) = D$
- Field 4: $F (|) 8 = F \Rightarrow$ Operation = OR
- Field 5: $F (\wedge) A = 5 \Rightarrow$ Operation = XOR
- Field 6: $F (&) ? = B \Rightarrow$ Operation = AND, thus $? = B$
- Field 7: $F (|) E = E \Rightarrow$ Operation = OR

Step 3a: Now that we know what the function, lets calculate the right input:

- Field 0: $? \wedge C = D \Rightarrow ? = 1$
- Field 1: $? \& F = E \Rightarrow ? = E$
- Field 2: $? \wedge 5 = A \Rightarrow ? = F$
- Field 3: $? \& D = D \Rightarrow ? = D$
- Field 4: $? | 8 = B \Rightarrow ? = B$ (or 3)
- Field 5: $? \wedge 5 = 3 \Rightarrow ? = 9$
- Field 6: $? \& B = 3 \Rightarrow ? = 3$
- Field 7: $? | E = F \Rightarrow ? = 1$

Step 3b: Now lets party: H(1EFDB931) = DEADB33F

So this concludes the basics of identifying operations on the single element. Now, let's look at bit shifting. As I mentioned before, I wanted to keep this simple. Subsequently, we will only operate on the byte level. Shifting is fairly simple to recognize by creating inputs that alternate elements. A good example input for this exercise is xF0F0. Now if our data is shifted to the right one byte or one byte to the left, we might get an output like this respectively:

R(xF0F0) = x0F0F

L(xF0F0) = x0F00

This is a good tactic, but there is something wrong here. Sometimes when shifts occur the lost byte might be rolled to the other end like so:

RO(xF0F0) =x 0F0F

LO(xF0F0) = x0F0F

Now we don't know which way the shift went, so we need to change it up by using another value in place of one of the 'F's:

RO(xA0F0) = x0A0F

LO(xA0F0) = x0F0A

See the difference :)

The next item we will discuss is transposition. Transposition is simply where input values are taken and then shuffled around in the output like so:

T(A00A) = 0AA0

Here we can't tell which one was which, so we need an extra test. In this test, we use all distinct input values in the following manner:

T(ABCD) = BCAD

Now we see that the C went to position 1 and A went to position 3.

Sorry no examples for the transposition stuff. I am running short on time.

In summary, when identifying operations performed on inputs, try using all 1's and 0's (e.g. F's and 0's) first to help deduce any obvious possibilities. If a input is suspected to be shifted, modify the values to be alternating such as 'A0F0F0' with at least one unique non-zero value so you can identify where the shift began and in which direction it occurs. Finally, if there is a transposition being used, attempt to identify which values are being swapped by using distinct values and observe where they end up, or by isolating values with 0's and the position being tested. Additionally, always keep a paper and pen handy for taking notes.

Challenge 2: SQL Injection

As shameful as it is for us to admit, we did not get this challenge. We developed a case of tunnel vision and attacked the wrong page. Doh!

Challenge 3: Password Brute Force

In this challenge, we needed to find a password that would let us log in to the remote system. From the challenge description, we knew that the password was all lower case and only alpha-numeric. We were also given a hash that matched the password and an algorithm to create the hash. The algorithm would use the secret key to initialize the RC4 algorithm, and then we would use resulting key stream to encrypt the string "::ShmooCon2008::" Finally, we were given a known password and its corresponding hash.

Taking this one step at a time, first we wrote a basic script that perform the given encryption scheme, then we used the known password and hash to check our output. If it is right we are rockin' and rollin', and we can move onto a brute force attack with a dictionary (or a few). When that fails, we can perform manual dictionary attacks with known terms from the movie, Wargames ;). The last part was easy, because with iPython, the script can be run and the functions remain in the environment. Thus, we can call the functions manually, yay!

Table 2 shows the code we used in an iPython environment to find the password.

Table 2: Python Code used in Challenge 4 to test lower case and leet speak passwords

```
#challenge4.py
from Crypto.Cipher import ARC4
# process the returned hash and make it look like
# the hash we are comparing
def get_hex(l):
    h = [hex(ord(i))[2:] for i in l]
    h_str = ''
    for i in h:
        if len(i) != 2:
            h_str += '0'+i
        else: h_str+= i
    return h_str.upper()

# convert words to leet speak
def convert_to_leet(s):
    convert = {'s':'5', 'l':'1', 'a':'4', 'e':'3', 'o':'0', 't':'7'}
    new = []
    for i in s:
        if convert.has_key(i): new.append(convert[i])
        else: new.append(i)
    return "".join(new)

# check the given password
# with both the actual password
# and the leet speak password
def check_pass(i):
    passwd(i)
    previous = i
    i = convert_to_leet(i)
    if i == previous: return
    passwd(i)

# creates a password hash and then
# checks the hash
def passwd(i):
    hash = "AD36C308E6B5FE5F1BF2FC18A890267B"
    secret = "":ShmooCon2008:"
    r = ARC4.new(i.lower())
    h = r.encrypt(secret)
    h_str = get_hex(h)
    print hash, h_str
    # quit if the value is found
    if hash == h_str:
        print "Match: %s"%i
        sys.exit()
    return h_str

# initial password list to use
f = "password.lst"

passwords = open(f, 'r').readlines()
hash = "AD36C308E6B5FE5F1BF2FC18A890267B"
# used to test the password brute forcer
#passwords = ["secret"]
#hash = "D70C8174EFCBB9E55DA589ECC3DECFF0D"

# check passwords in the list
for i in passwords:
    check_pass(i)
#winning password
#check_pass('joshua')
```

Basically, we called iPython challenge4.py, and it would use the supplied dictionary. Unfortunately, the password was not in any of the dictionaries we had, so we had to perform manual testing. On the iPython command prompt we simply had to do `check_pass('password')`, so we resorted back to my knowledge of Wargames (its in my Netflix queue ;) and guessed 'joshua.' In the `check_pass` function we perform 2 checks 1 if it is all lower case and another if it is all lower case in leet speak, and we nailed it on the first guess with 'j05hu4'.

Challenge 4: Proprietary Compression

In this challenge, we were given a file that was compressed with a proprietary algorithm, and we were given the algorithm to implement and uncompress the file. Then we had to md5 the file and that was the flag. Initially, we attempted this challenge in Java (why is beyond, all the cool kids were doing), but we were fail, and we resorted back to C (sweet sexy C). The compression algorithm was a compression algorithm to encode UTF-16 encoded characters. UTF-16 encoded characters have the property that they often share the same first byte, and only differ in the second byte. For instance, ABC would be encoded in UTF-16 as U+0041 U+0042 U+0043. The compression algorithm is as follows: the first byte is an unsigned char that contains the number of bytes to be decompressed. The next byte is the UTF-16 header, one char in length, that will prepend all the decompressed UTF-16 characters. The next byte is the second byte of the first UTF-16 character. The following bytes is the difference of the second byte of the previous UTF-16 character and the second byte of the current UTF-16 character. The following example comes from the documentation:

If "Scotch" was encoded the UTF-16 would look like:

```
U+0053 U+0063 U+006f U+0074 U+0063 U+0068
```

The sequence is six bytes long and the common most significant bits are 00h so the header becomes:

```
06 00
```

The encoded least significant bits would look like:

```
first byte: 53h
second byte:    53h - 63h = f0h
third byte:    63h - 6fh = f4h
fourth byte:   6fh - 74h = fbh
fifth byte:    74h - 63h = 11h
sixth byte:    63h - 68h = fbh
```

All together the encoded run of "Scotch" would look like:

```
06 00 53 f0 f4 fb 11 fb
```

Table 3: C Code used to read and decompress the file in Challenge 4

```
#include <stdio.h>
int main(int argc, char** argv){
FILE* inFile, *outFile;
```

```
char inChar1, inChar2, len, UTFheader;
unsigned char ulen;

inFile = fopen(argv[1], "r");
outFile = fopen(argv[2], "w");

while((len = fgetc(inFile)) != EOF) {
    ulen = (unsigned char) len;
    UTFheader = fgetc(inFile);
    inChar1 = fgetc(inFile);
    fputc(UTFheader, outFile);
    fputc(inChar1, outFile);
    ulen--;
    while(ulen > 0) {
        inChar2 = fgetc(inFile);
        fputc(UTFheader, outFile);
        fputc(inChar1 - inChar2, outFile);
        inChar1 = inChar1 - inChar2;
        ulen--;
    }
}
fclose(inFile);
fclose(outFile);
}

</code>
```

Challenge 5: Extract the Secret Picture

In this challenge, a file was embedded in a picture, and we needed to extract this file from the picture. The algorithm for the challenge was given, but since it was a .Net Application we decided to use .Net Reflector to see if the file was obfuscated and how it was making all its calls and such. Fortunately, there was another function in the application that is used to extract the file from the picture, which is shown in Figure 1.

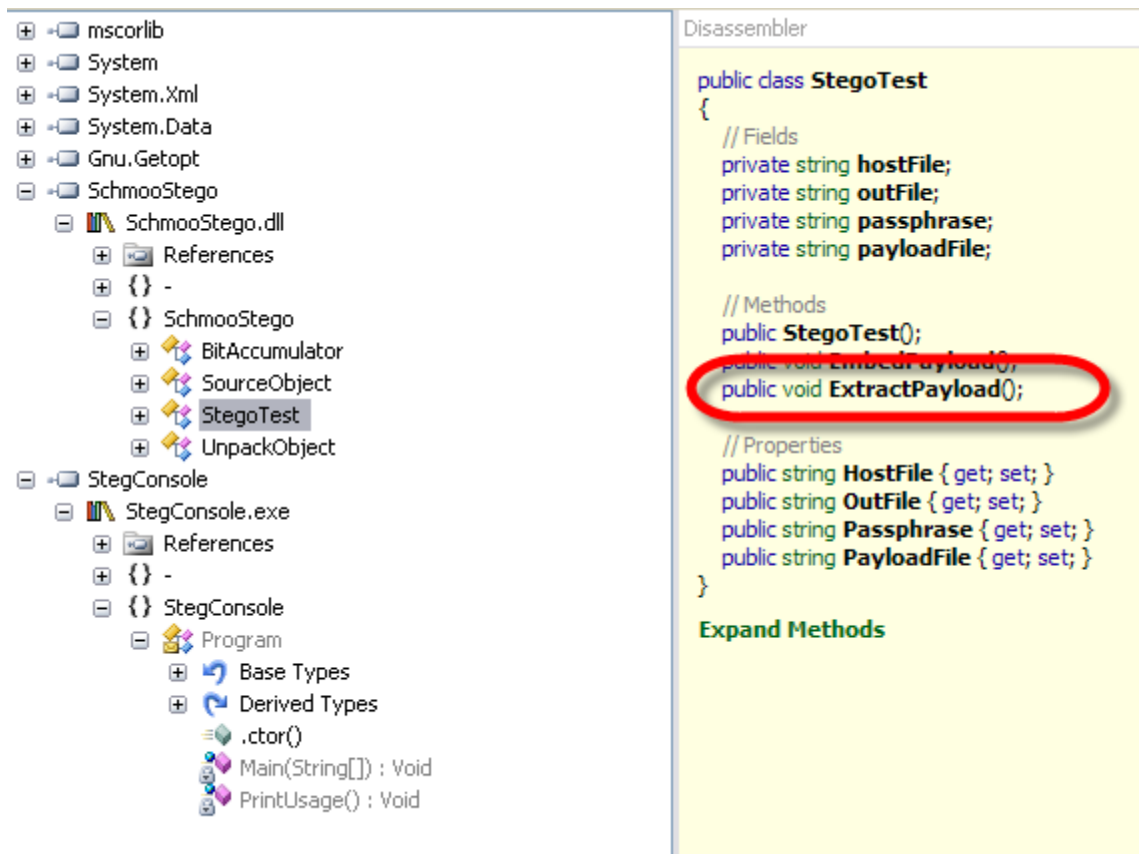


Figure 1: Shows the ExtractPayload method for the StegoTest Class using .NetReflector.

Once we identified this method, we looked where the EmbedPayload Method was being called from, and we used Reflexil to change that call to ExtractPayload. Figure 2 demonstrates the overall process.

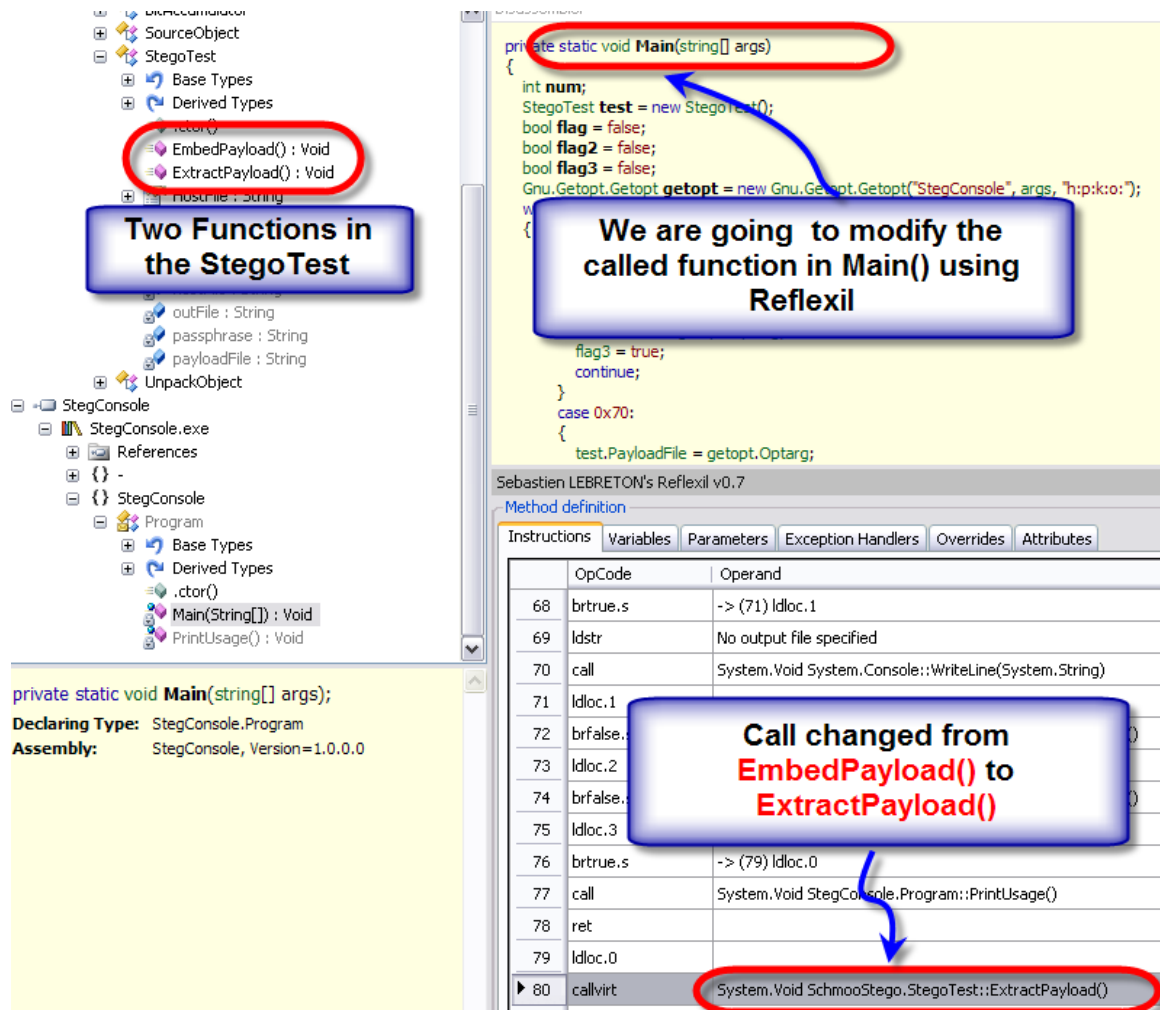


Figure 2: Shows the method call being changed to `ExtractPayload` in `Main()` using the Reflexil addin for .Net Reflector.

Starting from left, we identified the two methods in `StegoTest`, and then on the right, we found where the function call in `Main` was taking place, and swapped out the function calls. So, instead of calling `EmbedPayload`, the application will call `ExtractPayload`.

Challenge 6: Replaying RTP

In challenge, we opened up a TCP connection to the specified port 10006, and then listened on port 9000. Due to some pre-challenge festivities, we could not figure out how to get Wireshark to decode the RTP traffic, which is fairly trivial (See: <http://wiki.wireshark.org/rtpdump>). So, we ended up using `rtpools` to capture and replay the traffic. First, we used `rtp_dump` to capture traffic, then we used `rtp_play` to replay the traffic to get the solution. The following method was used to capture the traffic after performing the TCP connection:

```
rtpdump -F dump /9000 > challenge6.rtp
```

After we captured the RTP traffic, we created an SDP file for VLC client on Linux and used `rtp_play` to play the stream for the client. The SDP descriptor file (`challenge6.sdp`) had the following lines:

```
m=audio 5000 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
```

On the command line we enter the following command, but don't hit enter yet:

```
rtpplay -T -f challenge6.rtp 127.0.0.1/5000
```

Next, we open up VLC on a Linux client, and then we perform File->Quick Open File and select `challenge6.sdp`. Now we go back to the command line and hit enter. If VLC is not playing, just hit the play button and it should play. All of the above information was based off a very informative email on the [Ethereal –Users list](#), where the entire process for replaying an RTP stream is described.

Challenge 7: Blowfish client

We are fail.

Challenge 8: Tic-Tac-Toe

In this challenge, we needed to develop a client that would play Joshua autonomously and result in a win or a draw until Joshua gave up the code for this challenge. The resulting client was written in Python, and its development was broken into smaller stages.

Our Solution

In the first stage, we developed a client that would let us play Tic-Tac-Toe interactively with Joshua. This stage let us define all the necessary functions for creating and updating the board, mostly for debugging purposes. In the next phase, we would negotiate the first game, and then let the client and Joshua play each other. We moved onto the next phase after we identified and removed all the bugs for maintaining the game state. The final stage took the longest and this was focused on debugging our client's moves and correcting, so Joshua would not win. At this point, we had to switch from a recursive solution to an iterative (*e.g. infinite loop*), because the number of games would cause a stack overrun in Python. We knew the stack size could be adjusted, but we were pretty impatient and could not remember the process of adjusting it. For those interested in seeing our final solution, the code is shown in Appendix A.

A Better Solution

Someone brought up idea about piping one server's game to another, so the server in a sense is playing itself. We wish this would have been apparent to us sooner, because it would have saved us a lot of work. In theory (since we did not implement it), the client would broker a connection between the two Joshua servers, *favored* and *other* for the sake of explanation. The *favored* sever will be the one we want to win, and *other* server will be the one to play the opposing side. In order to get the two servers to take opposing sides, the client will just issue a new game request to the *other* server until it ends up

as the opposing player. Now all the client needs to do is pass the moves between the two connections. Doh!

Conclusion

So that's a wrap. We discussed how we tackled each challenge along with the tools employed. We talked about how to perform some basic differential analysis, and we also showed the code employed in the different challenges.

Acknowledgements

Special thanks to Applied Security for hosting HackIT 2.0 and then another giant thanks to the organizers and speakers of Shmoocon. We would also like to thank Nathan Sportsman for his review.

Appendix A: Code for an Autonomous Tic-Tac-Toe Client

```
import socket,sys

WINNING_CODE = ''
board = ['-'] * 10

MOVES = 0
LAST_MOVE = 0
PKT_TYPE = {
    0:'MSG_TYPE_NEWGAME',
    1:'MSG_TYPE_MOVE',
    2:'MSG_TYPE_ENDGAME',
    99:'MSG_TYPE_SHUTDOWNCODE',
    255:'MSG_TYPE_INVALID_MOVE'
}
WINNERS = {
    0:'PLAYER',
    1:'SERVER',
    2:'TIE'
}
MOVE = {
    'X':'MOVE_X',
    'O':'MOVE_O'
}

def new_game():
    s = socket.socket()
    s.connect(("victim.hackit",10008))
    pkt = s.recv(1024)
    print [i for i in pkt]
    return s,pkt

def play_game():
    s = socket.socket()
    s.connect(("victim.hackit",10008))
    pkt = s.recv(4)
    games = 0
    while True:
        t = interpret_pkt(pkt)
        start = int(ord(pkt[2]))
        if t == 0 and start == 0:
            pkt = x_play(s)
        elif t==0 and start == 1:
            pkt = o_play(s)
        elif t == 2:
            pkt = end_game(s,pkt)
            games+=1
        elif t == 99:
            winner(pkt,s)
        else:
            print "FAIL Invalid move!"
            sys.exit()

def x_play(s):
    global board;
    board = ['-'] * 10
    global MOVES
```

```

MOVES = 0

play = True
while play:
    pos = calculate_x()
    p = x_send_move(s, pos)
    t = interpret_pkt(p)
    if t == 1:
        spos = int(ord(p[3]))
        print "Server: ",[i for i in p]
        update_board(spos, 'O')
    elif t == 2:
        print 'End Game: ',[i for i in p]
        #play = end_game(s,p)
        return p
    elif t == 255:
        print "FAIL Invalid move!"
        sys.exit()

def o_play(s):
    global board;
    board = ['- ' for i in xrange(0,10)]
    global MOVES
    MOVES = 0

    pkt = s.recv(4)
    play = True
    p = pkt
    while play:
        spos = int(ord(p[3]))
        print "Server: ",[i for i in p]
        update_board(spos, 'X')
        pos = calculate_o()
        if pos == None or pos == 0:
            print "Maybe a Bug maybe not"
            print "pos is = " , pos
            p = s.recv(4)
            t = interpret_pkt(p)
            if t == 2:
                print 'End Game PKT: ',[i for i in p]
                #play = end_game(s,p)
                return p
            else:
                print "Dont know what to do her: ",[i for i in p]
        p = o_send_move(s, pos)
        t = interpret_pkt(p)
        if t == 2:
            print 'End Game PKT: ',[i for i in p]
            #play = end_game(s,p)
            return p
        elif t == 255:
            print "FAIL Invalid move!"
            sys.exit()

def end_game(s,pkt):
    print [i for i in pkt]
    winner = int(ord(pkt[2]))
    print "Winner is: ",winner
    if winner == 0:
        print "You are the Winner"
    elif winner == 2:
        print "It was a Tie!"
    else:

```

```

        print 'Server wins!!!'
        sys.exit();
    p = s.recv(4)
    #t = interpret_pkt(p)
    # check if shutdown!
    #print [i for i in p]
    #if t == 99: winner(s)
    ##Not shutdown restart game?
    #first = int(ord(p[2]))
    #if t == 0 and first == 0:
    #    x_play(s)
    #elif t != 0:
    #    print "Bug in end game"
    #    print [i for i in p]
    #    return None
    #else:
    #    p = s.recv(4)
    #    o_play(s,p)
    return p

def winner(p,s):
    np = s.recv(1024)
    p = p+np
    print [i for i in p]
    print "".join(p[2:])
    print [i for i in np]
    print "#####Done!"
    sys.exit();
    return None

def x_send_move(s, num):
    update_board(num, 'X')
    msg = '\x01'+'\x02'+chr(num)
    print "Client: ",[i for i in msg]
    s.send(msg)
    p = s.recv(4)
    return p

def interpret_pkt(pkt):
    print pkt
    t = int(ord(pkt[0]))
    size = int(ord(pkt[1]))
    #print 'Type = ',PKT_TYPE[t]
    #print 'Size = ',size
    if t == 0:
        move = int(ord(pkt[3]))
    elif t == 1:
        print "*** Recv'd a move"
    elif t == 2:
        print "EndGame"
    elif t == 99:
        print pkt[2:]
    elif t == 255:
        print "Bad Move!"
        print pkt
    return t

def get_move(pkt):
    move = int(ord(pkt[3]))
    return move

```

```

def set_winning_pkt(s):
    print "You are a winner"
    global WINNING_CODE
    WINNING_CODE = s.recv(1024)

def print_board():
    global board
    print ' %s | %s | %s'%(board[1],board[2],board[3])
    print '-----'
    print ' %s | %s | %s'%(board[4],board[5],board[6])
    print '-----'
    print ' %s | %s | %s'%(board[7],board[8],board[9])
    print '\n'

def o_send_move(s, num):
    update_board(num, 'O')
    msg = '\x01'+'\x02'+chr(num)
    print "Client: ",[i for i in msg]
    s.send(msg)
    p = s.recv(4)
    return p

def calculate_x():
    global MOVES;
    if MOVES == 0:
        MOVES += 1;
        return 1
    elif MOVES == 1:
        MOVES += 1
        if board[3] == 'O': return 9
        elif board[7] == 'O': return 3
        elif board[9] == 'O': return 7
        elif board[2] == 'O': return 5
        elif board[4] == 'O': return 5
        elif board[5] == 'O': return 9
        elif board[8] == 'O': return 5
        elif board[4] == 'O': return 5
        elif board[5] == 'O': return 9
        else: return find_open_square()
    else:
        #Playing defence here look for pairs of O's, or just play a square
        if board[3] == 'O' and board[5] == board[3] and board[7] == '-': return 7
        elif board[3] == 'O' and board[7] == board[3] and board[5] == '-': return 5
        elif board[5] == 'O' and board[5] == board[7] and board[3] == '-': return 3
        elif board[3] == 'O' and board[6] == board[3] and board[9] == '-': return 9
        elif board[3] == 'O' and board[9] == board[3] and board[6] == '-': return 6
        elif board[9] == 'O' and board[6] == board[9] and board[3] == '-': return 3
        elif board[2] == 'O' and board[2] == board[5] and board[8] == '-': return 8
        elif board[2] == 'O' and board[2] == board[8] and board[5] == '-': return 5
        elif board[5] == 'O' and board[5] == board[8] and board[2] == '-': return 2
        elif board[5] == 'O' and board[4] == board[5] and board[6] == '-': return 6
        elif board[4] == 'O' and board[4] == board[6] and board[5] == '-': return 5
        elif board[5] == 'O' and board[6] == board[5] and board[4] == '-': return 4
        elif board[7] == 'O' and board[7] == board[8] and board[9] == '-': return 9
        elif board[7] == 'O' and board[7] == board[9] and board[8] == '-': return 8
        elif board[8] == 'O' and board[8] == board[9] and board[7] == '-': return 7
        else: return find_open_square()

def update_board(pos, char):
    print 'Position: ',pos, "Char: ",char
    global board

```

```

if board[pos] != '-': print 'Invalid Move??'
board[pos] = char
print_board()

def find_open_square():
    global board
    for i in xrange(1,10):
        if board[i] == '-':return i
    print 'There is an ERRO!!!'
    return None

def calculate_o():
    global MOVES;

    if board[5] == 'X' and board[5] == board[9] and board[3] == '-' : return 3
    if board[5] == '-':return 5
    if block_x() != 0: return block_x()
    if x_corner() != 0: return x_corner()
    if x_caddy_corner() != 0: return x_caddy_corner()
    if MOVES == 0 and board[5] == 'X':
        MOVES += 1
        print 'X is in the middle'
        return 1

    if is_x_on_corner() and is_x_on_edge():
        if x_corner() != 0: return x_corner()
        return find_open_square()

    #Look for 2 Xs bordering the corners
    if x_border_corner() != 0:
        MOVES = 10
        #Now we just play defence
        return x_border_corner()
    #Look for 2 Xs bordering the corners
    if x_on_corners() != 0:
        MOVES = 10
        #Now we just play defence
        return x_on_corners()

    return find_open_square()

def block_x():
    if board[1] == 'X' and board[1] == board[3] and board[2] == '-': return 2
    elif board[1] == 'X' and board[1] == board[2] and board[3] == '-': return 3
    elif board[1] == 'X' and board[4] == board[1] and board[7] == '-': return 7
    elif board[1] == 'X' and board[1] == board[7] and board[4] == '-': return 4
    elif board[2] == 'X' and board[3] == board[2] and board[1] == '-': return 1
    elif board[3] == 'X' and board[5] == board[3] and board[7] == '-': return 7
    elif board[3] == 'X' and board[7] == board[3] and board[5] == '-': return 5
    elif board[5] == 'X' and board[5] == board[7] and board[3] == '-': return 3
    elif board[3] == 'X' and board[6] == board[3] and board[9] == '-': return 9
    elif board[3] == 'X' and board[9] == board[3] and board[6] == '-': return 6
    elif board[9] == 'X' and board[6] == board[9] and board[3] == '-': return 3
    elif board[2] == 'X' and board[2] == board[5] and board[8] == '-': return 8
    elif board[2] == 'X' and board[2] == board[8] and board[5] == '-': return 5
    elif board[5] == 'X' and board[5] == board[8] and board[2] == '-': return 2
    elif board[5] == 'X' and board[4] == board[5] and board[6] == '-': return 6
    elif board[4] == 'X' and board[4] == board[6] and board[5] == '-': return 5
    elif board[4] == 'X' and board[4] == board[7] and board[1] == '-': return 1
    elif board[5] == 'X' and board[6] == board[5] and board[4] == '-': return 4
    elif board[7] == 'X' and board[7] == board[8] and board[9] == '-': return 9
    elif board[7] == 'X' and board[7] == board[9] and board[8] == '-': return 8

```

```

elif board[8] == 'X' and board[8] == board[9] and board[7] == '-': return 7
return 0

def x_on_corners():
    if board[1] == 'X' and board[3] == board[1] and board[2] == '-': return 2
    elif board[1] == 'X' and board[7] == board[1] and board[4] == '-': return 4
    elif board[3] == 'X' and board[9] == board[3] and board[5] == '-': return 5
    elif board[7] == 'X' and board[9] == board[7] and board[8] == '-': return 8
    else: find_open_square()

def x_caddy_corner():
    if board[1] == 'X' and board[9] == board[1]: return find_open_edge()
    elif board[1] == 'X' and board[9] == board[1]: return find_open_edge()
    elif board[7] == 'X' and board[7] == board[3]: return find_open_edge()
    elif board[7] == 'X' and board[3] == board[7]: return find_open_edge()
    return 0

def x_corner():
    if board[1] == 'X' and board[9] == '-': return 9
    elif board[3] == 'X' and board[7] == '-': return 7
    elif board[7] == 'X' and board[3] == '-': return 3
    elif board[9] == 'X' and board[1] == '-': return 1
    else: return 0

def find_open_edge():
    if board[2] == '-': return 2
    if board[4] == '-': return 4
    if board[6] == '-': return 6
    if board[8] == '-': return 8
    return 0

def x_border_corner():
    if board[2] == 'X' and board[4] == 'X': return 1
    if board[2] == 'X' and board[6] == 'X': return 3
    if board[8] == 'X' and board[4] == 'X': return 7
    if board[8] == 'X' and board[6] == 'X': return 9
    return 0

def is_x_on_corner():
    return board[1] == 'X' or board[3] == 'X' or board[7] == 'X' or board[9] == 'X'

def is_x_on_edge():
    return board[2] == 'X' or board[4] == 'X' or board[6] == 'X' or board[8] == 'X'

s,p = new_game()
print [i for i in p]
interpret_pkt(p)
print_board()

```