# Exercise 5 – Network Protocol Exploitation

**Deadline: 12.06.2013 15:00**

## Introduction

You are the CTO of a very promising startup. Your startup idea is to create the very best file transfer application in the world. You've spent the past half year thinking of a name to your application, which left you with little time to do any coding. The chosen name was eSquirrel™. You decided to hire the least expensive coder that was available online. To your amazement, the coder finished the entire client-server framework in 1 day, during which he took a 2 hour chicken-tikka break.

You became suspicious and decided to evaluate the quality of the product given to you. Since less than 50$ were paid, you were not able to convince your Indian coder to give you the actual source code, only the compiled executables.

In this exercise you are given a (bad) client server implementation. Download `client.exe` and `server.exe` from the website and play around with them. Make sure you understand what these programs do (from a user standpoint). In the following sections you will analyze the underlined network protocols and find weaknesses in the programs.

## Configuration

This exercise is mostly a research exercise. You will need to combine your knowledge in Wireshark, IDA and OllyDbg in order to understand the inner-workings of the programs and the underlined network protocol. It's a good idea to combine these tools together. Some tasks will be much easier to solve via Wireshark than to try to reverse engineer the code, some tasks are the opposite. Some tasks are only solvable by using the 3 tools together.

During the exercise you will sniff communication between the client and the server. Because you can't sniff on your loopback adapter, you'll need to create a setup with at least two (maybe virtual) computers to work with. Here are 2 possible configurations:

1. Using a virtual machine, run the server on the host/guest while executing the client on the guest/host. Use Wireshark on one of them and sniff the virtual networking device.
2. Run the client on one physical computer and the server on another computer in the same subnet. Use Wireshark on one of them and sniff the actual network adapter you use to connect to this network. This is the easiest option when working in the lab.
3. Use two virtual machines on the same physical computer.

## Implementation

All code in this exercise can be written in either C or Python. Each has its advantages and you may alternate between the two in the different questions. If you chose to implement something in Python, submit a `.py` file and not a `.c` file as specified in the question. Obviously your executable becomes a `.py` instead of `.exe`. Python implementation must run in the lab under Python 2.7 (Install it if you need it).

We decided to add Python as a representative of scripting languages. We believe it's quite a handy tool to add to your belt. Sadly, we are unable to test more than two programming languages, thus code which is not C or Python will not be accepted.

**Note**

This exercise is longer and arguably harder than all previous exercises. It can be viewed as a summary of most material covered in class so far. You'll need to use all your previously acquired skills, and take it up a notch. The exercise involves a great deal of research. Don't be alarmed if you spend a lot of time without any output - if the research part is done well, the coding part should go smoothly.

Please start early. As always, extensions will only be given with due cause.

The weight of this exercise in your final grade is greater than the rest of the exercises.

Good luck :)

## Part 1 – The Protocol (15 Points)

In this part you'll need to describe the full protocol of eSquirrel™ - the best file sharing application in the world. The protocol consists of 2 UDP packets and 4 TCP packets. For each of these packets you'll need to specify the following:

- UDP / TCP
- Source and destination ports
- Packet's content

Where "packet's content" describes the actual bytes of data. Here you need to explain only the bytes that belong to the application level protocol, and not bytes that belong to lower level protocols (TCP, IP and so on). Specify a table with 3 columns: field name, size and field meaning. An example for such table that describes some (made up) packet is as follows:

| Field Name | Size (in bytes) | Field Meaning |
|---|---|---|
| **Magic** | 4 | `Constant value. Always 0x12345678` |
| **SizeOfString** | 2 | `Specified the size of the following string` |
| **String** | `SizeOfString` | `An informative ASCII text string` |

Submit the protocol description in a file named `protocol.txt`.

## Part 2 – The Server's ID (15 Points)

You've probably realized by now that the server generates an ID upon execution. This ID is used by the protocol to uniquely identify the different servers in the network.

Investigate the ID generation algorithm. Do you think there's a problem with the way the server ID is generated? If you answered no, skip to Part 3. If you answered yes – explain in the README.

Note that the code was compiled with optimizations. This has several impacts. For example – some parameters passed to functions might be passed through registers, and a function with no obvious input parameters may actually have some. Also, this is probably the first time you'll encounter IDA being wrong. Don't forget to correct its mistakes. Some useful features for this part are the ability to edit struct types, to redefine variable types (if the detected type is wrong) and to redefine the stack frame (if it wasn't created correctly by IDA). Read about these features of IDA. Correctly using them will greatly improve your experience in this part.

Create a file called `parseSID.c` with the following usage:

```
parseSID.exe <SID>
```

In this file you'll implement a Windows Console Application. This application will receive a server ID as input, and output to the user, in human readable strings (when possible) all the information it can gather regarding that server.

Your implementation may assume valid input. Explain in the README how the SID is generated.

## Part 3 – File Access Weakness (20 Points)

There's a weakness in the client/server suite which allows access to unauthorized files on the server (the weakness does not involve a buffer overflow). Describe the weakness in the README. Create a client named `peek.c` with the following usage:

```
peek.exe <SID> <path to file on server>
```

The code will retrieve that file to the current directory in peek's computer.

Hint: you were asked to implement your own client for a reason. The weakness might not be visible when using the regular client.

## Part 4 – Hijacking (20 Points)

There's a possible hijacking problem in the protocol. A malicious node in the network can send specially crafted packets under some circumstances in order to hijack a connection between a client and a server.

In your README you'll need to explain this problem thoroughly:

- Which side can be faked, the client or the server?
- How can this be done?
- Detail all malicious packets that are sent in order to hijack the session.

Write code named `cyanide.c` with the following usage:

```
cyanide.exe <SID>
```

When running, the code should hijack all the connections made to the given server and act as the server (meaning that all type of packets should be processed in the same way as they are processed in the server, and you should return data from the hijacking computer). You can assume there are no two simultaneous requests to the server at the same time.

The hijacker must run forever, and as long as it's running it must continue hijacking all connections to the given SID.

If you had to analyze specific packet fields in order to complete the hijacking process, explain in the README how these fields are generated.

Your `cyanide.exe` must perform its task successfully when executed on any machine in the local network except for machines with running eSquirrel™ servers.

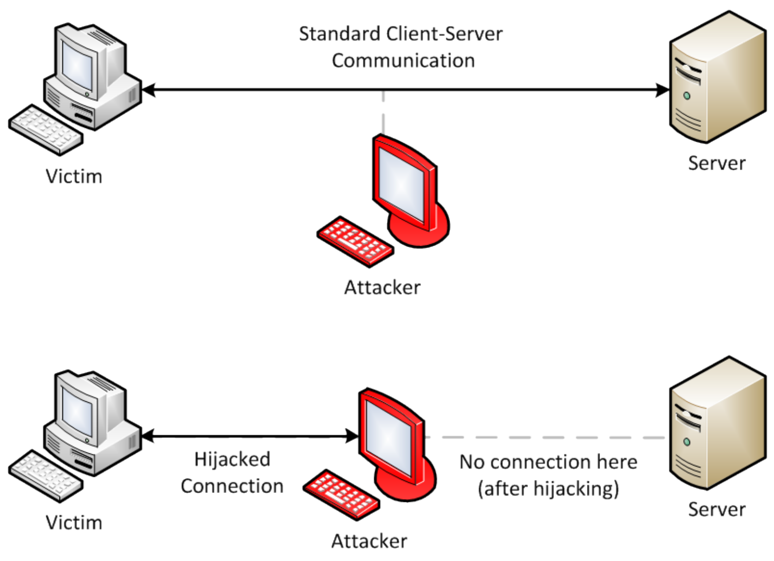
Standard Client-Server Communication
Victim
Server
Attacker
Victim
Hijacked Connection
Attacker
No connection here (after hijacking)
Server

Figure 1 – (Top) Before connection hijacking; (Bottom) after connection hijacking

## Part 5 – Man in the Middle (30 Points)

In the previous part, we stole all communication from the legitimate server. In this part we'd like to perform something a bit more subtle. You are required to implement a "Man in the Middle" attack. Create a file named `mitm.c` with the following usage:

```
mitm.exe <SID>
```

This code will hijack connections to the given server and will pass requests and responses between the client and the real server (act as a proxy). The hijacker will act as follows:

- If a `.jpg` file is requested by the client (`*.jpg`), a file with the same name but different contents will be provided instead of the requested file. The new file will be an image to your liking. Any image will do, as long as it's less than 1MB. Your image may be inlined in the code or read from an external file (the presubmit script allows a file called `pic.jpg`).
- All other communications must not be altered.
- Bonus (3 points): If a directory listing is requested, the attacker should return the original list without any files or directories containing "`$sys$`" in their name. Please specify in the README if you implemented this bonus.

Again, your `mitm.exe` must perform its task successfully when executed on any machine in the local network except for machines with running eSquirrel™ servers.

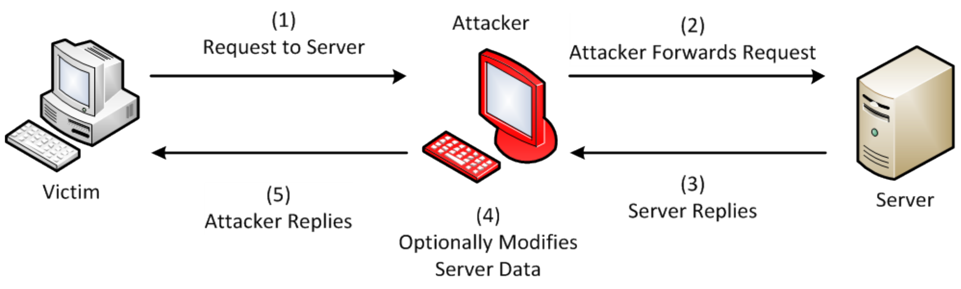

Figure 2 - The MITM attack

## Bonus 1 – (Simple) Buffer Overflow (8 points)

There's a buffer overflow in the client. Find the vulnerability; specify what you've found in the README. Exploit the vulnerability in order to display the message box from exercise 2 and resume normal execution. Submit code named `evil_server.c`. Your code will implement a server in the system, which exploits the vulnerability when talking to a client.

## Bonus 2 – (Awesome) Buffer Overflow (25 points)

Create a file named `awesome_evil_server.c` with the usage:

```
awesome_evil_server.exe <SID>
```

The code will do the following:

- Hijack connections made to the given server ID.
- Upon client connection, the awesome server will use the vulnerability from bonus 1 to run code on the client.
- Your shellcode should change the client's code in memory to allow a new type of packet from the server to the client. This special packet will include a string of command to execute on the client using `system()` or `WinExec()` API calls.
- The client does not need to send a reply back with the output of the command.

Specify in the README file the exact format of your new packet, the details of the vulnerability and explain the implementation of your exploit.

If you implemented this bonus, there's no need to implement bonus 1. If both bonus 1 and 2 are submitted, you will not be graded on bonus 1.

## Bonus 3 – Custom Wireshark Dissector (10 points)

The protocol eSquirrel™ is a proprietary protocol. Because of that, Wireshark doesn't know how to dissect it, and displays the entire application level as "Data" instead of meaningful fields. Extend Wireshark so that it'll understand the eSquirrel™ protocol.

There are many ways to do the above. Choose whichever one you want and specify what you did in the README. Submit a file named `dissector.tar.gz` that contains all the files relevant to your implementation. Supply compilation instructions in the README.

## Bonus 4 – MITM with On-The-Fly .EXE Manipulation and Network Propagation (50 points)

Submit a file named `coolest_mitm_on_earth.tar.gz` which contains files that do the following:

- Perform a MITM attack for connections made to all servers in your local network on all network interfaces.
- If a client requested an executable file (.EXE) under your hijacked connection, you should return a dynamically created .EXE file which does the following:
    - Spawns a hidden executable and executes it.
    - Executes the original file.
- The hidden executable should be selectable by the user ("executable payload" below). The hijacker has the following usage:

  ```
  coolest_hijacker.exe [<executable payload>]
  ```

  If no argument is given the hijacker will use a default executable payload, as explained below.

- The hijacker should predict the size of the new, modified executable so it will be able to replace the size of the file in a directory listing given to the victim client.
- The default payload acts as follows:
    - If no server is running on the current computer, the logic of `coolest_hijacker.exe` will be executed (we state here logic instead of the actual executable since you need to hide and one option to do so is to avoid being an executable).
    - If a server is running, the logic of `coolest_hijacker.exe` will not be executed.
    - If all servers stop running, the logic of `coolest_hijacker.exe` is executed.
    - You cannot rely on the executable name in order to identify the server. Use a more robust mechanism (such as binary signatures).
    - The above behavior should be maintained across reboots of the machine.
    - The above code must not appear as a process in the process list.

Specify in the README all the parts of your implementation and how they interact.

## Submission

All solutions should work under the Windows machines in Ross 19.

The following files should be included in the submitted tarball file:

- `protocol.txt`
- `parseSID.c / parseSID.py`
- `peek.c / peek.py`
- `cyanide.c / cyanide.py`
- `mitm.c / mitm.py`
- `pic.jpg`
- `evil_server.c / evil_server.py` (bonus 1 only)
- `awesome_evil_server.c / awesome_evil_server.py` (bonus 2 only)
- `dissector.tar.gz` (bonus 3 only)
- `coolest_mitm_on_earth.tar.gz` (bonus 4 only)
- README – Formatted according to the course guidelines.

A single TAR file named `ex5.tar` should be submitted. Make sure you run the pre-submit script before submitting the exercise.

Good luck!