Analysis and reverse-engineering of a multiplayer online game

Informe final

Joel Puig Rubio

Resum– Els jocs en línia són una molèstia pels esforços de preservació de jocs. L'ús de protocols de capa d'aplicació propietaris, combinat amb la seva naturalesa en constant evolució, els fa notòriament difícils d'analitzar. Com a resultat, aquests jocs sovint no es poden salvaguardar en una capacitat jugable després del final de la seva vida, cosa que pot significar la seva mort a mesura que la lògica i els actius del costat del servidor deixen de ser disponibles. L'objectiu del projecte és mostrar maneres d'abordar aquest problema agafant un joc existent i creant una implementació del servidor que podria substituir el servidor de jocs original.

Paraules clau– Enginyeria inversa, joc en línia, emulació de servidor

Abstract– Online games are a thorn in game preservationist's side. The use of proprietary application layer protocols combined with their ever-evolving nature makes them notoriously hard to pin down. As a result, these games often fail to be saved in a playable capacity after their end-of-life, which can spell their death as server-side logic and assets become unavailable. The goal of the project is to show ways to tackle this problem by taking an existing game and creating a server-side implementation that could replace the original game server.

Keywords– Reverse-engineering, online game, server emulation

1 INTRODUCTION

1.1 Motivation

As the internet became a more pervasive aspect of our lives, the gaming industry realized that it had the potential to revolutionize the way players interact with games. This development spawned an array of new genres, some of which work solely because of this online component. However, this proved a challenge for preservation, as these games are now dependent on a third-party service that could cease its operation. Unfortunately, it is not common practice for the publisher or developer of these games to release the server-side component in any capacity, rendering these games partially or completely unplayable.

• E-mail de contacte: joel.puig.rubio@gmail.com
• Menció realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Marcel Vilalta i Soler (Departament d'Enginyeria de la Informació i de les Comunicacions)
• Curs 2021/22

1.2 Scope

This project aims to tackle this issue by analyzing an existing game and implementing a new server that emulates the real one. To this end, we applied reverse-engineering techniques to unravel the inner workings of its proprietary application layer protocol and design a compatible server.

Our target is Tanki Online [1], a game by Alternativa Games where you control a tank and the goal is to destroy those of other players. The game is based on Adobe Flash and uses the ActionScript 3 programming language. We also know that it uses a proprietary protocol on top of TCP for networking.

2 OBJECTIVES

The main objectives of this project were:

- Reverse-engineer and document the application layer network protocol used by the game
- Develop a compatible game server without or minimally altering the client

We hoped this would allow us to gain much knowledge about how game servers and protocols work in practice.

In addition, we had the following secondary objectives to fulfill if time allowed for it:

- Develop tools to aid with the reverse-engineering efforts
- Apply common programming paradigms in the development of the server
- Measure its performance under load, and analyze the bottlenecks, if any

2.1 State of the art

Over the years, there have been many attempts to reverse-engineer game servers for all kinds of games to varying degrees of success. I would say this primarily comes down to two factors:

- Complexity: Referring to how complex a game is in terms of features, or how difficult it is to reverse-engineer
- Availability: Referring to the sources from which we can recover information relating to the game, such as the original server, packet dumps from other users or binaries and other assets from the client

Games programmed in a scripted language or one that compiles to an intermediate language (IL), such as ActionScript, Java or C are easier to reverse-engineer than one that has been compiled to a native executable. This is because the compiled executable still contains much more debugging information than otherwise.

Generally, to be able to create a server emulator for a game, we need some way to examine the packets sent to and from the original game server. There are several ways to accomplish this, which may be chosen depending on convenience or availability.

2.1.1 Prior art

I would like to briefly highlight some projects I have come across during my research that have tackled game server emulation.

Raise the Empires This is a community project to preserve the Flash game Empires & Allies by Zynga [2]. This game used to be hosted within Facebook, as an in-platform game, meaning that your Facebook profile would be tied to in-game progress. Like Tanki Online, this game was also built in ActionScript 3. However, its development has been impaired by a low degree of availability: the original server is long gone and assets for the client disappeared with it. Thankfully, one of the core contributors was able to recover many assets from an old computer's cache. For some of the more critical remaining assets, the team attempted to recreate them from whatever information they had available.

Wiimmfi Another community project, with the goal to bring back the so-called Nintendo Wi-Fi Connection (WFC) services, which a number of Nintendo games depended on for online play. Its development started with Mario Kart Wii and now supports more than 500 games [3]. Unlike the previous project, this one did not face availability issues as the developer started development before the shutdown of the services and had the foresight to save packet dumps for future reference.

2.1.2 Analysis through packet sniffing

The most obvious approach one may take to do this is to sniff packets to and from the server. This can be accomplished by an application such as Wireshark [4] or though another application that acts as a man-in-the-middle (MITM) [5].

Wireshark allows the user to capture and analyze in detail any packets going through a given network adapter [6]. It also can drill down into a number of well-known application layer protocols such as HTTP. This is a very useful tool if our game uses a proprietary protocol over TCP or UDP.

In addition, any captured packets can be saved for later as a packet dump file. Packet dumps are invaluable to be able to perform the reverse-engineering process even if the original servers have gone offline. Moreover, packet dumps can be performed by people who do not know reverse engineering capabilities themselves and share them with someone who does.

It is important to notice that encryption of the packets can difficult our task to sniff them [7]. Some games use this as a dissuasive tactic against reverse-engineering attempts, but may use a static encryption key or may share it with the server over plaintext, which may allow the rest of the conversation to be decrypted at a later date. However, if a protocol like TLS is employed, we may be in for trouble. In the following section we will go over an alternative method to sniff packets in the case of encryption.

2.1.3 Code patching

A possible way to circumvent the problem with encrypted protocols is to identify where packet encoding and decoding occurs in the game client and patch the game to dump plaintext packets to a file [7]. This can be accomplished in two different ways, statically or at runtime. However, the most common is to use find this routines using static analysis and patching the game's binaries.

2.1.4 Static analysis

Static analysis may be used in combination many other techniques. For example, when combined with packet sniffing, to understand the encryption used by a game. However, pure static analysis may also be used to figure out what the game expects the conversation with a potential game server to look like. This can be a good recourse when a live game server is no longer available, with certain caveats:

- Server-side behaviors: Depending on the degree to which a game is server authoritative there may be entire behaviors that are impossible to reproduce accurately because they were handled entirely server-side. If client/server authoritative were a scale, a purely client authoritative game would send the state of the game over to the server, which would simply relay it

to other clients. On the other hand, a purely server authoritative game server would take care of everything and send commands to the client with what to display or do next. Most games sit somewhere in-between the two.

- Resources loaded "on-the-fly": Some games may load required resources over the network, these can become lost if the server goes offline. Although some games may cache these to disk in a way that they can be recovered [8], this is not always the case.

2.1.5 Dynamic analysis

Dynamic analysis is the process by which code is analyzed while it is being run. Although there are many processes that fit the description of "dynamic analysis", for the purposes of reverse-engineering a game, this will often refer to the use of a debugger. This may be used in situations where static analysis is difficult because of code being hard to read, whether deliberately, by result of obfuscation, or simply due to the complexity of certain routines.

However, many games employ the use of anti-cheat code that often includes anti-tamper mechanisms to hinder the use of debuggers. While these can mostly be defeated, it may hamper novice attempts at reverse-engineering.

3 METHODOLOGY

3.1 Approach

In the planning stage, we determined three crucial tasks required to establish a path towards the completion of the project's main objectives:

- Establish methods to extract information from the game
- Determine the process by which information will be analyzed
- Decide on the technology used to implement the server

I will begin by explaining how I tackled the task of extracting of information from the game. This was necessary for the next point, in which I talk about processing this data to obtain meaningful information necessary for understanding the game's network protocol. Finally, I will discuss the technology that has been chosen for the development of the server emulator.

3.1.1 Information extraction: Obtaining packet dumps

Using Wireshark, we were able to look at Tanki Online's network traffic. This involved some HTTP requests to gather some configuration options about the client, including a list of online game servers. The client then picks one of them and initiates a TCP connection.

Once the connection is established to the game server, we can see some printable strings containing bits of information about the environment of the client. Unfortunately, it is not possible to make sense of much else after that. Suspecting encryption, we looked at the disassembly of the game client and located some encryption routines. Fig. 1 shows the extent of the communication which could be decoded before encryption kicks in.

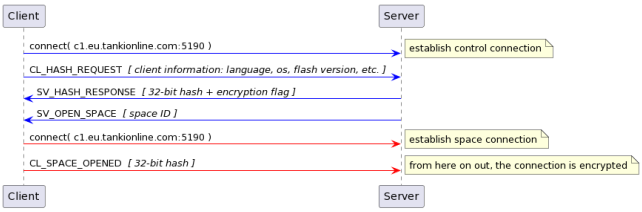

Fig. 1: Handshake between Tanki Online client and server

The most immediate thought was to patch the code to dump unencrypted packets to disk to examine what they look like. However, for security reasons, Adobe Flash does not directly allow writing files to disk except in very specific circumstances. Instead, Flash developers are encouraged to make use of so called "shared objects" [9], which are reminiscent of the more modern HTML5 local storage API.

Frustrated by this, we opted to print packets to the console. However, Flash does not have a console, and we had to switch to the debugging version of Flash Player, as by default, the trace() function, which is the one that permits us to obtain the packet dumps, redirects nowhere.

Thankfully, switching to a debugging version of Flash Player is easy enough. The Flash Player debug projector is available as a standalone binary. However, we also need to create a settings file at "`~/mm.cfg`" [10] to actually enable trace output. Once that is done, trace output is written to a text file, located at "`%AppData%/Macromedia/Flash Player/Logs/flashlog.txt`" on Windows.

3.1.2 Information analysis: Inspecting and parsing packets

With a proper way of extracting packets from the game, analysis of the proprietary protocol began. This process consisted of a combination of looking at the individual packets, one at a time, and cross-referencing them with the decompiled code to figure out their structure and what their purpose in the exchange is.

This required the creation of scripts that ingest extracted packets, then parsed and annotate them into a more human-readable form. As we got further into the exchange, and document it, we started to form a sort of mental picture on how the protocol works. In addition, it gave us some clues on how we would write our own implementation.

3.1.3 Server emulator technology

As for the server implementation, we settled on a solution using Java and the Netty framework. Java was chosen because of my existing familiarity with the language. In addition, I thought its static type system would ease development and code readability compared to a dynamically typed language. Due to the unfamiliar nature of the systems and objects we would deal with, I believed a dynamically typed language could result in more instances where we would lose track of types or confuse them, leading to a slower and more frustrating development process. The Netty framework provides an event-driven abstraction layer [11] around socket primitives and provides robust handling of connec-

tions so we can focus on the definition of the service itself. It defines a set of useful and easy to use interfaces that satisfy many common needs of a UDP or TCP server.

3.2 Implementation

Earlier in the development process, we were able to reach the game's login screen by feeding the client packets dumped using our existing tools. Our next task is to inspect the contents of these dumped packets to learn what they do and build these programmatically instead. However, before that, we must implement the encryption used in the protocol in our server reimplementation.

3.2.1 Protocol encryption

During our work of the project, time we switched a flag in the control command *SV_HASH_RESPONSE*, which would disable encryption in the client. Although implementing this serves no practical purpose given the existence of this flag, we must if we strive to create a faithful implementation.

The encryption routines are found in a script named *XorBasedProtectionContext*, and as the name implies it involves the use of a lot of bitwise-XOR operations. The use of XOR is common in many ciphers, so this does not give us much information.

Cipher initialisation The cipher takes two parameters, a 32-byte long buffer named *hash* and a 64-bit integer named *spaceId*. During the initialisation steps, these are both combined through the use of XOR into a 32-bit integer key, named *initialSeed*.

1. Initially, *initialSeed* is set to 0.
2. Each byte of the *hash* buffer is XORed into *initialSeed*. After each operation, the resulting value is stored in *initialSeed*.
3. Next, *spaceId* is unpacked into 8 individual bytes, starting from the MSB and XORed into *initialSeed* in the same way as before.
4. After all these operations, if the value of *initialSeed* is greater than 127, we subtract 256 from *initialSeed*.

After this, *initialSeed* is used to initialized two 32-bit integer arrays with 8 elements each, named *serverSequence* and *clientSequence* respectively:

1. Set a counter *n* at 0. Repeat the following steps eight times.
2. Shift the value of *n* three bits to the left. This gives you the *n*-th power of 2. Store this value in a register.
3. XOR the value of *initialSeed* with the value in the register. Store the result in the n-th slot in the *serverSequence* array.
4. XOR the value of *initialSeed* with the value in the register, then XOR the result with 87. Store the result in the *clientSequence* array.
5. Increase the counter *n* by 1.

Finally, we initialize two 32-bit integer values to 0. They will be used during the ciphering processes and are named *clientSelector* and *serverSelector* respectively. If we are the client, we must use *clientSequence* and *clientSelector* to encipher, and *serverSequence* and *serverSelector* to decipher, or vice-versa.

Encrypting Repeat these steps for each byte we want to encipher:

1. Store the value of the byte we want to encipher in register A.
2. Read the slot in the sequence array pointed to by the selector. Store it in register B.
3. Copy the value of register A into the slot in the sequence array pointed to by the selector.
4. Perform a bitwise AND of register A and 7. Then perform a XOR of the selector and the result of the previous operation. Store the result back into the selector.
5. XOR the value of register A with register B. Store the result back into A.
6. Register A contains our enciphered byte.

Decrypting Repeat these steps for each byte we want to decipher:

1. Store the value of the byte we want to decipher in register A.
2. Read the slot in the sequence array pointed to by the selector. Store it in register B.
3. XOR the value of register A with register B. Store the result back into A.
4. Copy the value of register A into the slot in the sequence array pointed to by the selector.
5. Perform a bitwise AND of register B and 7. Store the result into the selector.
6. Register A contains our deciphered byte.

The values of *spaceId* and *hash* are sent over to the client in cleartext, so this cipher is vulnerable to man-in-the-middle attacks. Due to that, it seems more likely that this cipher is an attempt to dissuade and provide security through obscurity rather than an attempt at data protection. Although it is a bit concerning, given that user credentials are transmitted through this protocol, so a determined attacker could steal credentials of users in the same network.

We were not able to determine what kind of cipher this was based on. Thankfully, since the code was nicely readable it was possible to simply rewrite it in Java and slot it into our server.

3.2.2 Space prototypes

You may have seen these mentioned throughout this report but they have never been clearly defined. Spaces are confined environments that the server puts the client into, they contain a series of *game objects*, each of which can have *models* tied to them. Models are a series of remote procedures, they can be server procedures that the client can call or vice-versa. Each *space*, *game object* and *model* have a numerical id, for models it is a hardcoded id that lets both client and server refer to the same interfaces. Spaces are themselves a game object, the so called *root object* and their id is 0. They also each have a definite lifetime: creation, initialization and destruction.

In the server they are implemented as two different kinds of objects: *SpacePrototype* and *Space*, the former describes how the specific *space* behaves while the latter contains information about a *Space* instance. It may be possible to refactor this into a single class, but this is just the way it has been done for now.

3.2.3 Encoding of native protocol types using a type adapter pattern

The protocol establishes a set of encodings for primitive types which are combined to build more complex encoders. While the Flash client addresses this by calling individual primitive encoders from code-generated *codecs*, we opted to address this with a composable type adapter system. This idea came from a JSON encoding library I have used in the past called Moshi [12], developed by Square. The final result is a combination of Moshi's implementation with some extensions specific to satisfy the requirements of the Alternativa protocol. In my personal opinion, this resulted in a rather elegant solution.

The building stone for each adapter resides in a class named *BufferAdapter*, not unlike *JsonAdapter* in Moshi, which declares an interface for all of its implementations:

```
public abstract class BufferAdapter<T> {

    public abstract T fromBuffer(ProtocolBuffer buf);

    public abstract void toBuffer(ProtocolBuffer buf, T value);

    public BufferAdapter<T> nonNull() {
        return new NonNullBufferAdapter<>(this);
    }

    public BufferAdapter<T> optional() {
        return new OptionalBufferAdapter<>(this);
    }

    interface Factory {
        BufferAdapter<?> create(Type type, ObjectEncoder encoder);
    }
}
```

Fig. 2: Code for *BufferAdapter*

The main two operations of a *BufferAdapter* are *fromBuffer* and *toBuffer*: the first takes *ProtocolBuffer*, a protocol primitive which contains an OptionalMap and a byte buffer and decodes into an object of an arbitrary type, the second does the reverse operation, it takes a *ProtocolBuffer* and an object which to be encoded into the buffer

The remaining two are a bit more interesting, and are related to the domain-specific "extensions" I mentioned earlier. *nonNull* wraps the given adapter into an adapter that behaves identically, except if it is given a null value in *toBuffer*, in which case it throws an exception. This is related to *OptionalMap* functionality for types not strictly marked as nullable as they also need to be marked as such in the *OptionalMap* for the decoding process to succeed. *optional* wraps the given adapter into an adapter that behaves identically, except that it emits an *OptionalMap* bit for types marked as nullable.

The *Factory* interface, as its name implies, allows the creation of *BufferAdapter* factories for specific types and is used by *ObjectEncoder* which itself holds a registry of all available factories and a *BufferAdapter* cache for performance.

```
public static final BufferAdapter.Factory FACTORY = new BufferAdapter.Factory() {
  @Override
  public BufferAdapter<?> create(Type type, ObjectEncoder encoder) {
    if (type == boolean.class || type == Boolean.class) return BOOLEAN_BUFFER_ADAPTER.nonNull();
    if (type == byte.class || type == Byte.class) return BYTE_BUFFER_ADAPTER.nonNull();
    if (type == double.class || type == Double.class) return DOUBLE_BUFFER_ADAPTER.nonNull();
    if (type == float.class || type == Float.class) return FLOAT_BUFFER_ADAPTER.nonNull();
    if (type == int.class || type == Integer.class) return INTEGER_BUFFER_ADAPTER.nonNull();
    if (type == long.class || type == Long.class) return LONG_BUFFER_ADAPTER.nonNull();
    if (type == short.class || type == Short.class) return SHORT_BUFFER_ADAPTER.nonNull();
    if (type == String.class) return STRING_BUFFER_ADAPTER.nonNull();
    return null;
  }
};
```

Fig. 3: *BufferAdapter* factory used for primitive types

Another factory exists which crafts *BufferAdapters* for arbitrary POJOs (Plain Old Java Objects) using these adapters, there also exist specific *BufferAdapters* for generic Java *List* and *Map* types.

3.2.4 Models and remote procedure calls

As we previously mentioned, models define a set of remote procedure call interfaces. The type adapter system fits right in here, as it allows us to implement them transparently as regular Java interfaces.

```
public class ChatModelServer extends Model {

    public ChatModelServer(Space space) {
        super(space);
    }

    @ModelId(6683616035809206555L)
    public void changeChannel(String channel) {
        ChatMessage welcome = new ChatMessage();
        welcome.messageType = 1;
        welcome.text = String.format("Welcome to #%s!", channel);
        ChatModelClient chat = space.getModel(ChatModelClient.class);
        chat.showMessages(Collections.singletonList(welcome));
    }
```

Fig. 4: Server-side model implementation (Java)

As displayed in Fig. 4, *ChatModelServer* defines a method called *changeChannel* that receives a parameter of type *String*, and in the body of the method, it takes a client-side model from the current space and calls the method *showMessages* with a parameter of type *List* on it.

The following interface defines *ChatModelClient*.

```
public interface ChatModelClient {

    @ModelId(4202027557179282961L)
    void showMessages(List<ChatMessage> messages);
}
```

Fig. 5: *ChatModelClient* interface

As for its implementation, there is none. Not on the server, of course. Any calls to this interface are dynamically

proxied, its arguments serialized into a remote-procedure call compatible with the Alternativa protocol and automagically sent over to the client.

If we turn to the decompilation of the client, this is the actual code the server is calling:

```
171     public function showMessages(param1:Vector.<ChatMessage>) : void
172     {
173        var _loc2_:ChatMessage = null;
174        var _loc3_:String = null;
175        var _loc4_:int = 0;
176        var _loc5_:BattleChatLink = null;
177        var _loc6_:BattleInfoData = null;
178        for each(_loc2_ in param1)
179        {
180           _loc3_ = _loc2_.text;
181           if(this.antiFloodEnabled)
182           {
183              Antiflood.getMessageKeys(_loc3_,true);
184           }
185           clientLog.log(LOG_CHANNEL_NAME,"showMessages : %1",_loc3_);
186           this.htmlFlag = false;
```

Fig. 6: Client-side model implementation (AS3)

As we can see, this provides pretty seamless interoperability between ActionScript 3 and Java code.

4 RESULTS

4.1 Testing game features in the client

Let us review which features we were able to introduce into our server implementation so far. While defining our goals for our project, we ruled out getting in-game as it is too much work in terms of replicating subsystems in the server: match creation, matchmaking, physics simulation, etc. Most of which would fall out of scope of our project. However, we can log-in and do several things around the lobby. One of which is being able to have a chat with other people, as demonstrated below.

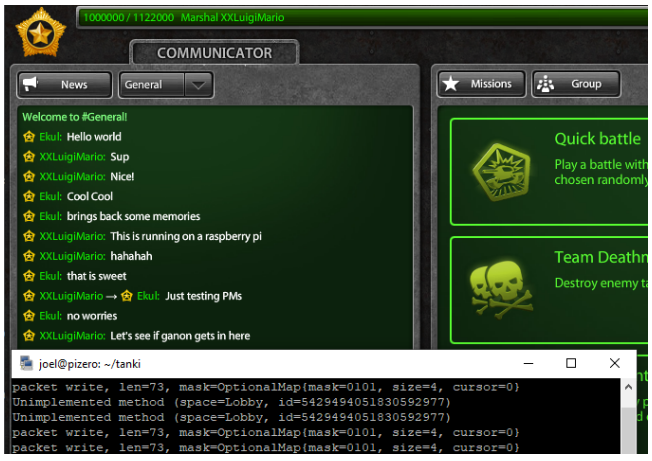

Fig. 7: Demonstration of the chat working with two people

In addition to messaging, the game has chat commands available to administrators and I implemented a couple for further testing.

- */addcry*: Adds in-game currency (crystals) to a player's profile
- */addexp*: Adds in-game experience to a player's profile

```
1  public class AddCry extends Command {
2
3     public AddCry() {
4        super("addcry");
5     }
6
7     @Override
8     public void handle(LobbyUser user, String[] args) {
9        int crystals = Integer.parseInt(args[0]);
10       user.addCrystals(crystals);
11       user.sendMessage("Your wish is my command");
12    }
13 }
```

Fig. 8: Example chat command implementation

4.2 Test suite

Some things like *OptionalMap* were tricky to implement because of the convoluted encoding algorithm, involving a great number of bitwise operations. For this reason, we created a few unit tests to ensure that our understandings were correct and there were no flaws in our implementation.

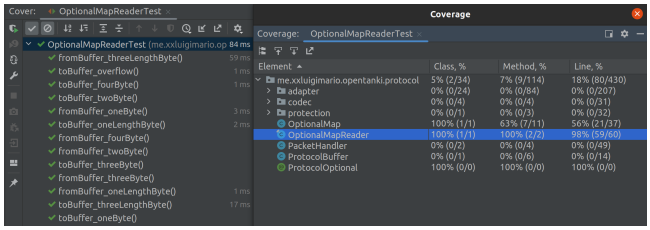

Fig. 9: Coverage statistics for *OptionalMapReaderTest*

As you can observe in Fig. 9, *OptionalMapReaderTest* brings *OptionalMapReader* to 100% code coverage. Although IntelliJ claims 98% line coverage, that is because one line is effectively unreachable and contains an assertion.

5 CONCLUSION

Our main goal was to take an existing game and reverse engineer its server protocol, and I believe we have been able to do that. We started from nothing and built a server that is able to hold a session with the game as if it were the official server, without any modification done to the client. Although it was not an absolute requirement, we also implemented the cipher used by the game protocol, so even inspecting the traffic with adequate tools you would be hard-pressed to find a difference against a regular game session.

We also created convenient tooling that can compile viable documentation of the game protocol from its binaries, this speeds up development and provides invaluable information, as binary protocols such as this one do not allow for any flexibility in its encoding. One mistake will render the rest of the message unreadable, leaving the two parties unable to communicate.

The server does not currently implement all subsystems required for the game to progress to an in-game match, but that is a matter of building on top of the framework we have created and is left as future work, and I personally found this project motivating enough to keep working on it and see this through.

6 ACKNOWLEDGEMENTS

First and foremost, I would like to thank my tutor, Marcel Vilalta i Soler, for his efforts and guidance, which ensured this project was able to fulfill its end goals within the designated time frame. In addition, I would like to acknowledge the support and feedback of my peers at BlueMaxima's Project in this endeavour which helped in shaping this project.

REFERENCES

[1] Alternativa Games, "Tanki Online". Accessed on: Mar. 6, 2022. [Online] Available: https://alternativa.games/games/tanki-online-2/

[2] AcidCaos et al., "Raise The Empires". Accessed on: Jun. 26, 2022. [Online] Available: https://github.com/AcidCaos/raisetheempires

[3] Wiimm et al. "Wiimmfi Main Page". Accessed on: Jun. 26, 2022. [Online] Available: https://wiimmfi.de/

[4] F. Faessler, "Information Gathering / Recon - Pwn Adventure 3". Accessed on: Apr. 10, 2022. [Online] Available: https://youtu.be/pzM4o6qxssk

[5] F. Faessler, "Developing a TCP Network Proxy - Pwn Adventure 3. Accessed on: Apr. 10, 2022. [Online] Available: https://youtu.be/iApNzWZG-10

[6] R. Sharpe, E. Warnicke and U. Lamping, "Wireshark User's Guide". Accessed on: Apr. 10, 2022. [Online] Available: https://www.wireshark.org/docs/wsug_html_chunked/

[7] Manfred, "Twenty Years of MMORPG Hacking: Better Graphics, Same Exploits" in: DEFCON 25, Las Vegas, United States, Jul. 2017

[8] BlueMaxima's Flashpoint, "Recovering Files from Browser Cache". Accessed on: Apr. 10, 2022. [Online] Available: https://bluemaxima.org/flashpoint/datahub/Recovering_Files_from_Browser_Cache

[9] Adobe, "SharedObject - Adobe ActionScript® 3 (AS3) API Reference". Accessed on: Apr. 10, 2022. [Online] Available: https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/SharedObject.html

[10] Adobe, "Configure the debugger version of Flash Player". Accessed on: Apr. 10, 2022. [Online] Available: https://web.archive.org/web/20210216051441/https://helpx.adobe.com/flash-player/kb/configure-debugger-version-flash-player.html

[11] Netty project, "Netty: Home". Accessed on: Apr. 10, 2022. [Online] Available: https://netty.io/

[12] Square et al., "A modern JSON library for Kotlin and Java.". Accessed on: Jun. 17, 2022. [Online] Available: https://github.com/square/moshi