Inspecting and Attacking the Citrix ICA Protocol using Canape

James Forshaw and Michael Jordon
whitepapers@contextis.com

Date: 16 March 2012

Contents

Introduction

Canape is a new network protocol analysis and manipulation tool for Windows which aims to reduces the amount of work required during a security review to assess an arbitrary protocol. It is designed to act in a similar fashion to pre-existing Web application testing tools such as CAT[i] and Fiddler[ii], providing an interface to capture, manipulate and then replay network traffic in any protocol, not just HTTP.

This whitepaper outlines how to use the tool to develop a framework for manipulating the Citrix Independent Computing Architecture (ICA) protocol. ICA is a proprietary networking protocol used by Citrix to provide remote application and desktop functionality for clients.

This protocol has been chosen because it is a complex binary protocol, something that Canape was developed to manipulate and it does not seem to have had significant amounts of security research aimed at it. Documentation for the protocol is scarce, and even Wireshark does not come with a dissector for ICA.

By the end of the whitepaper, the goal is to give the reader a better understanding of the ICA protocol itself and to give a suitable example for demonstrating the flexibility of the Canape tool for security testing and research.

Canape can be downloaded from Context website at http://canape.contextis.com[iii].

Canape Overview

Canape is a network testing tool for arbitrary protocols, but specifically designed for binary ones. It contains built in functionality to implement standard network proxies and provide the user the ability to capture and modify traffic to and from a server. The core can be extended through multiple programming languages including C# and Python, to parse any protocol as required thereby creating custom proxies tailored to the testing. It works at the network application layer supporting both TCP and UDP connections through port forwarding or by implementing a SOCKS or HTTP proxy. It does not capture data at the Ethernet, IP or TCP layers directly.

Its main strength is reducing the amount of development effort usually associated with effectively testing a new protocol. By providing a common mechanism to capture and manipulate traffic, it aims to allow the security researcher to only develop the minimal amount of code for the truly bespoke aspects of a protocol.

Developing a Canape Project

Canape groups the resources required to analyse and manipulate a protocol into a single project, similar in many respects to that used in an Integrated Development Environment (IDE) such as Visual Studio or Eclipse. The project might contain resources such as:

- Networking services, for example a SOCKS or HTTP proxy
- Directed network graphs defining the data flow and state model of the protocol
- User developed basic parsers
- Custom script code to parse more complex traffic, or to manipulate traffic in specific ways
- Captured data such as logs of packets
- Test harnesses, used to develop and test parsing code in isolation from a network connection.

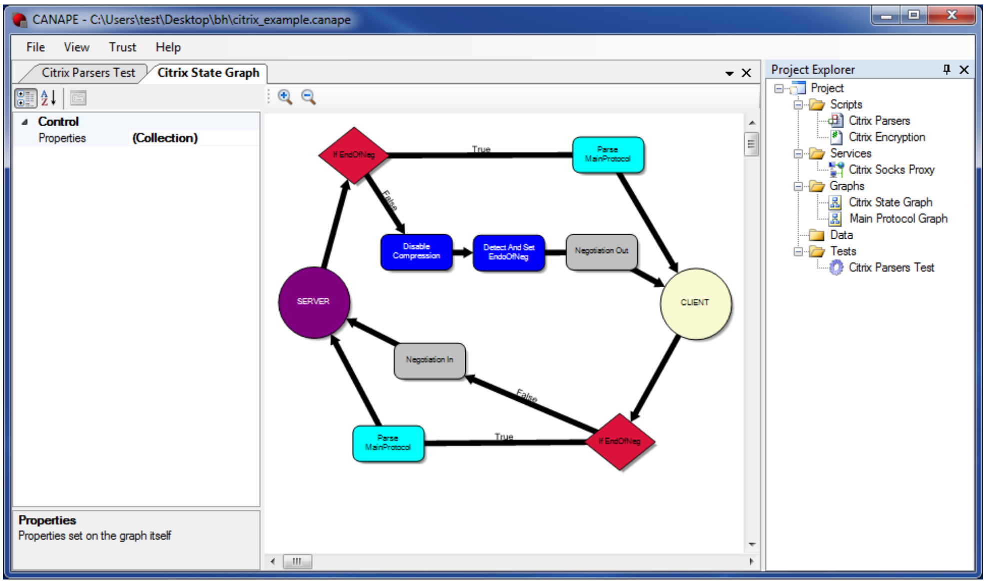


Figure 1 - Screenshot Showing Example Project

All project resources are saved into a single file, by default with the '.canape' extension. This whitepaper is accompanied by an example project to parse basic Citrix ICA protocol traffic; subsequent sections will refer to this project to reduce repetition.

Initial Traffic Capture

The first step in analysing a bespoke network protocol is capturing some example traffic; in order to do this using Canape a mechanism is required to force the traffic through a configured proxy. This can be done in a number of different ways, however some are more flexible than others. The following list shows some example approaches to getting traffic into Canape, in order of preference:

1. Configure the application to use a SOCKS or HTTP proxy
2. Use a third party tool (such as FreeCAP[iv]) to convert an application into using SOCKS.
3. Configure the application to go to a fixed IP address and port, and then use a fixed proxy.
4. If the application looks up network destinations through DNS add an entry to the 'hosts' file to redirect it to a fixed IP address.
5. In the case of enclosed devices (i.e. mobile phones) then a fake DNS server (which is supported in Canape itself) can be used to redirect the traffic to Canape.

Fortunately in Citrix it is possible to configure a SOCKS proxy for use when connecting to a server as show in the following client configuration file.

```
[WFClient]
Version=2
TcpBrowserAddress=1.1.1.1
ICASOCKSProtocolVersion=0
ICASOCKSProxyHost=127.0.0.1
ICASOCKSProxyPortNumber=1080

[ApplicationServers]
test=
[test]
TransportDriver=TCP/IP
WinStationDriver=ICA 3.0
DesiredHRES=800
DesiredVRES=600
DesiredColor=8
Address=1.1.1.1
```

Figure 2 - Example ICA File with Proxy Configuration Highlighted

To use this to capture traffic a SOCKS proxy can be created in Canape and started it up. By default the proxy will capture all outgoing and incoming packets through the proxy and display them in a packet log as shown:

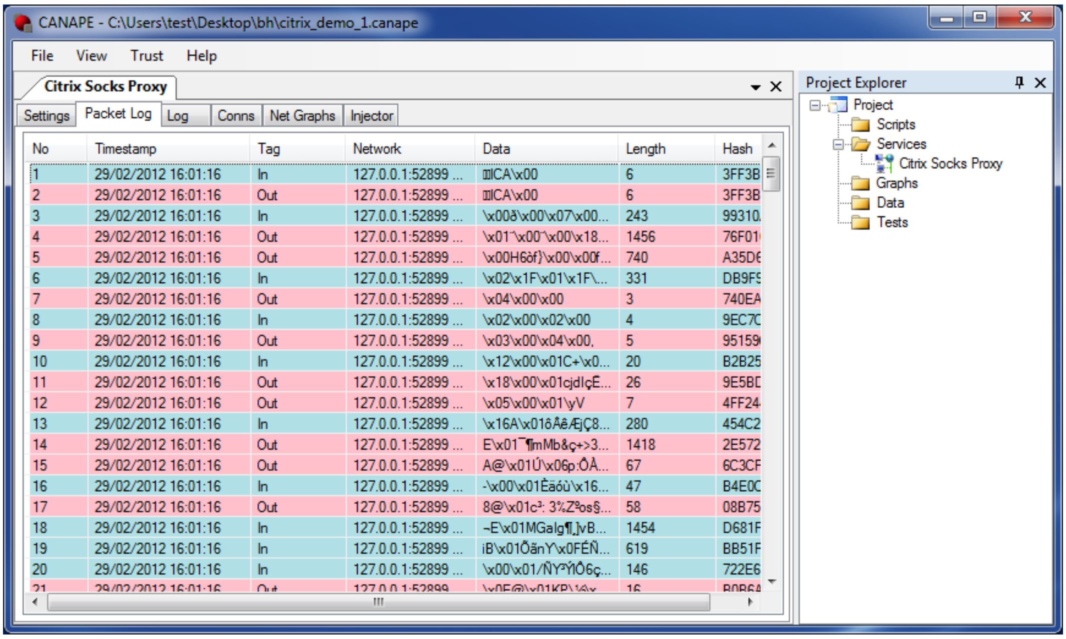


Figure 3 - Traffic Captured Through SOCKS Proxy

By double clicking individual packets it is possible to inspect each entry in greater detail. These packets can also be copied and pasted to other parts of the application as required, for example it is possible to copy packets into a test harness to aid in the development of custom parsers.

Through inspection of the ICA traffic it becomes clear that there are three phases to the protocol. First is a simple 'hello' identification phase, it starts with the server sending an ICA magic string (as show in Figure 4), the client will then respond back with the same value.

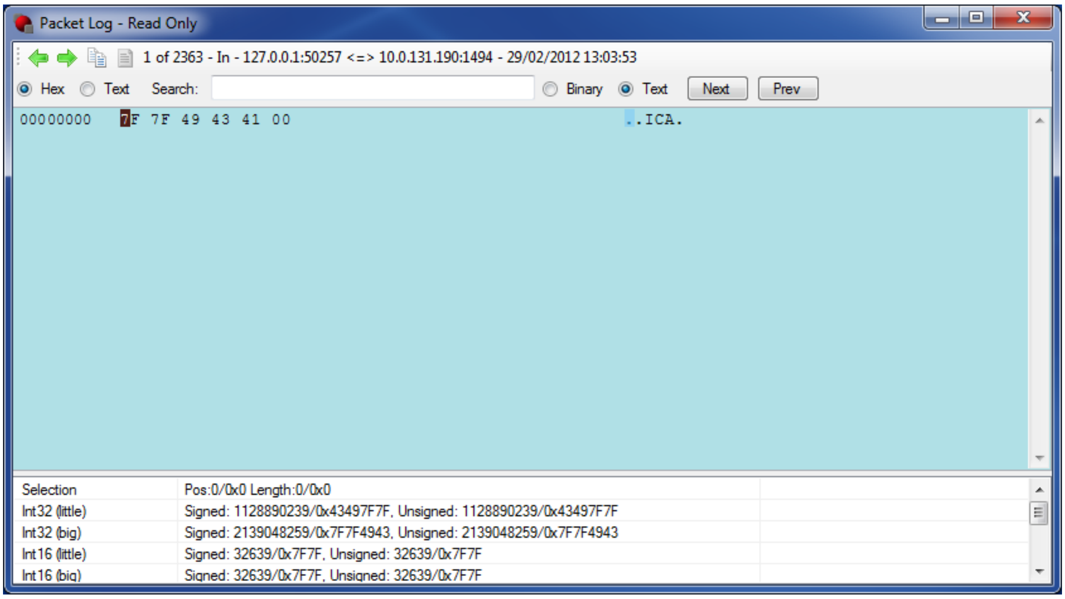


Figure 4 - ICA Magic String

After the magic strings have been passed the protocol enters a negotiation phase where the features of ICA are agreed. Each packet in the negotiation starts with a single byte representing the type of the packet. The next two bytes represent the length of the following data in little endian format (which is somewhat more unusual for network protocols). The negotiation is completed when a packet of type 4 is sent from the client.

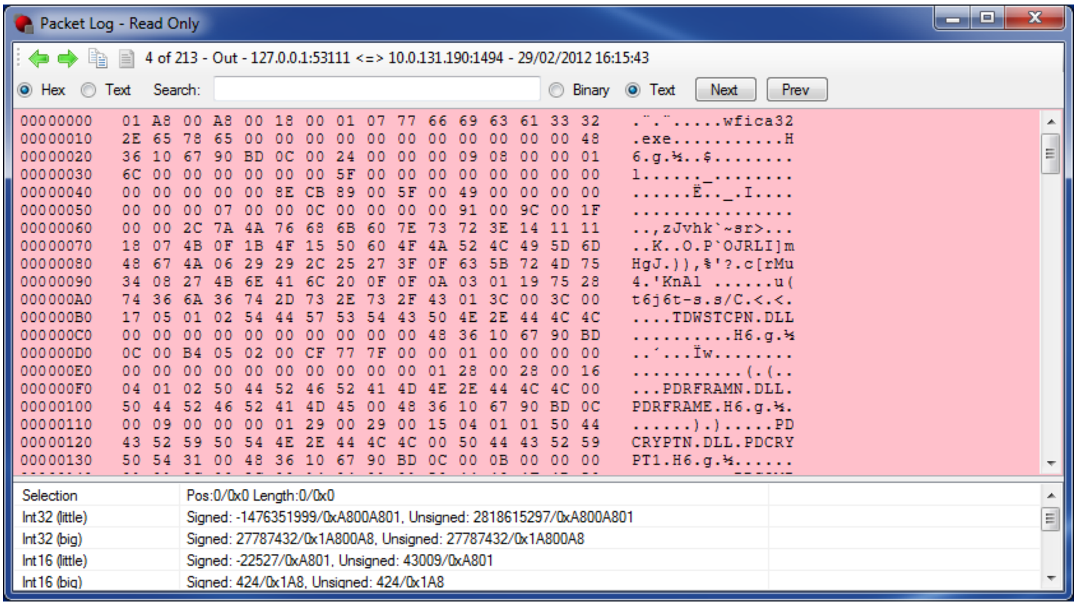


Figure 5 - Example Negotiation Packet

The final phase, which will be referred to as the 'main' protocol, now begins. Each packet is again fairly simple on the outside. Each 'frame' of the protocol starts with a 12 bit little-endian length field, followed by a 4 bit set of flags. This is followed by the number of bytes indicated in the length field.

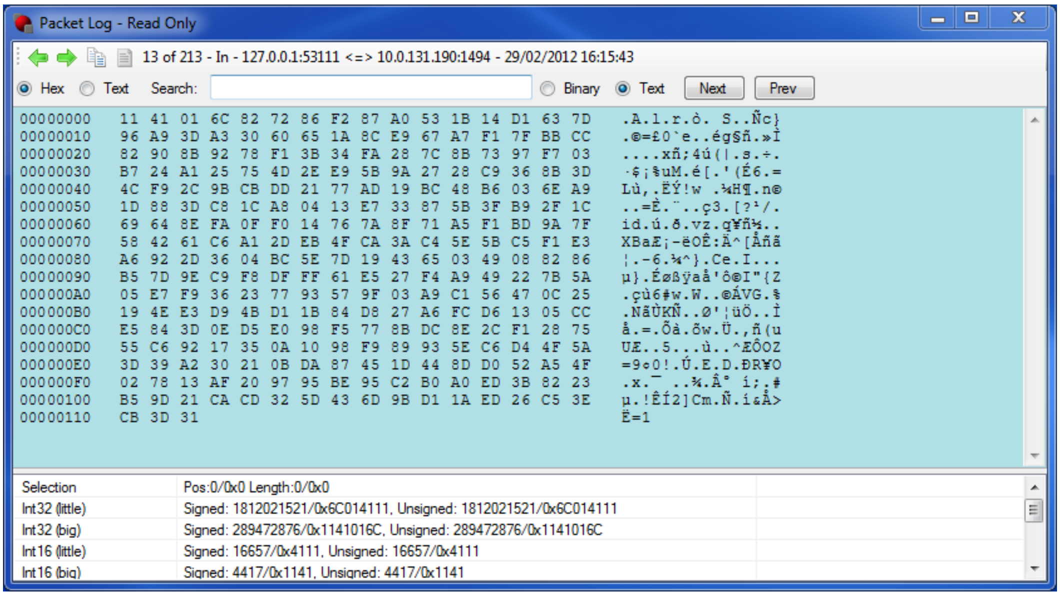


Figure 6 - Example Main Protocol Packet

Unfortunately there is now a problem, other than the initial length field and flags the rest of the packet seems to be encrypted, or at least encoded. By default Citrix client and servers employ 'Basic' encryption, this will need to be removed before the main protocol can be attacked. First, Canape needs to be configured to handle the three protocol phases so that specific parsing can be applied at the appropriate phase.

Modelling the Protocol State

In order to model the protocol state Canape provides a directed graph editor to represent what is termed a 'Net Graph' in the tool. These graphs serve two functions in Canape; firstly they provide the ability to model data flow. Each node on the graph represents some discrete function, for example parsing of a particular protocol or causing a packet to be logged (Canape only logs packets at points you explicitly tell it to). The other purpose is to model state transitions, a state value can be set which reflects where in the protocol the connection currently is, then simple decision nodes can be used to send packets for different types of processing.

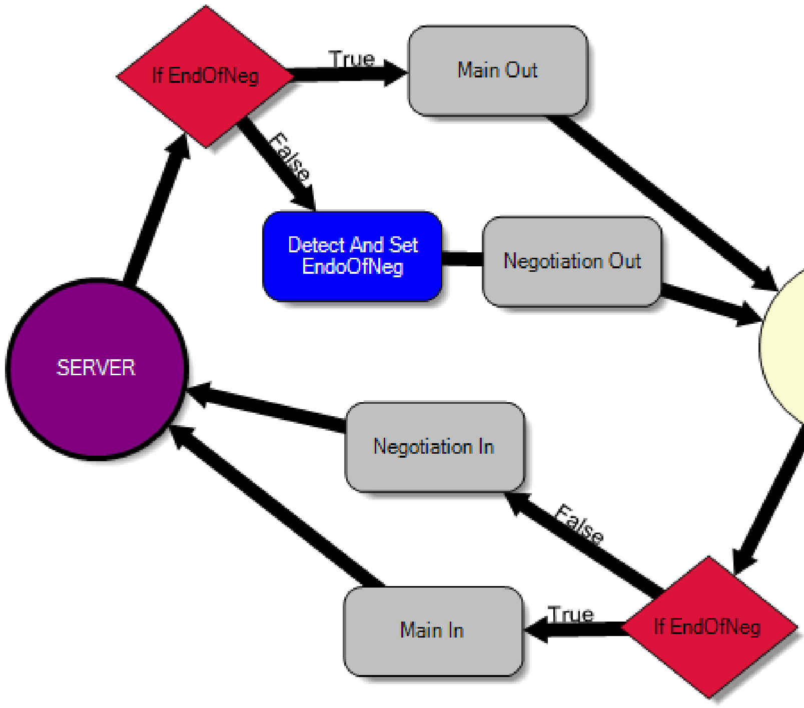


Figure 7 - Simple State Diagram for Citrix

Figure 7 shows the basic state diagram developed for the Citrix protocol. The initial 'hello' and negotiation phases have been merged as distinguishing between them provides little benefit. The large 'SERVER' node represents the location in the graph that packets coming from the client enter the graph, these packets then flow along the edges and are affected by the other nodes until it reaches the 'CLIENT' node. The rational for the naming convention is the 'SERVER' is in this case bound to the listening TCP socket server in the proxy, while the 'CLIENT' is bound to the TCP client connecting over the network to the real server.

The grey nodes represent logging elements, any packet which traverses one of these nodes is automatically logged to the packet log shown in Figure 3. The rhombus nodes are the decision elements, if the current state is set to "EndOfNeg" then packets are sent through the 'main' protocol parser (which in this example just means the packet is logged) otherwise it is sent through as a negotiation packet.

The final blue node is the mechanism through which the state change is produced. This node is configured to wait for the type 4 packet already described, at which point the state is changed to indicate the end of negotiation.

Parsing the Main Protocol

Now that the protocol phases are separated out the main protocol can be parsed. As the framing of this is a length/data based protocol, it is possible to do everything in Canape's built-in parser editor.

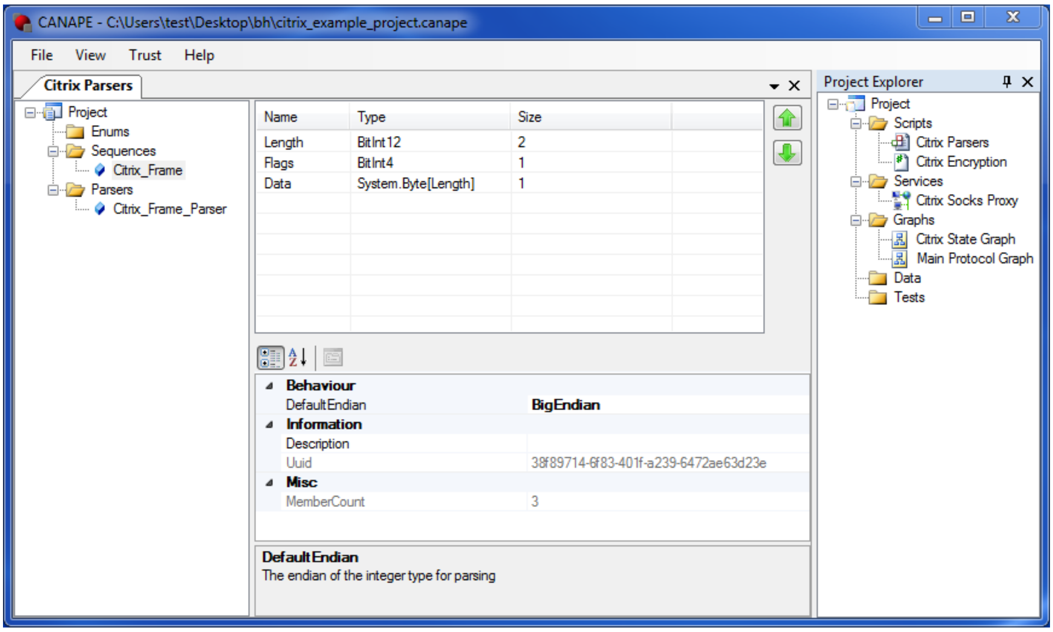


Figure 8 - Main Protocol Parser

Figure 8 shows the developed parser structure. It consists of a sequence of values and a parsing wrapper. This can then be added to the graph as a 'Dynamic' node. Once introduced, logged packets change from the previous raw binary data into a tree structure as show on the left-hand side of Figure 9.

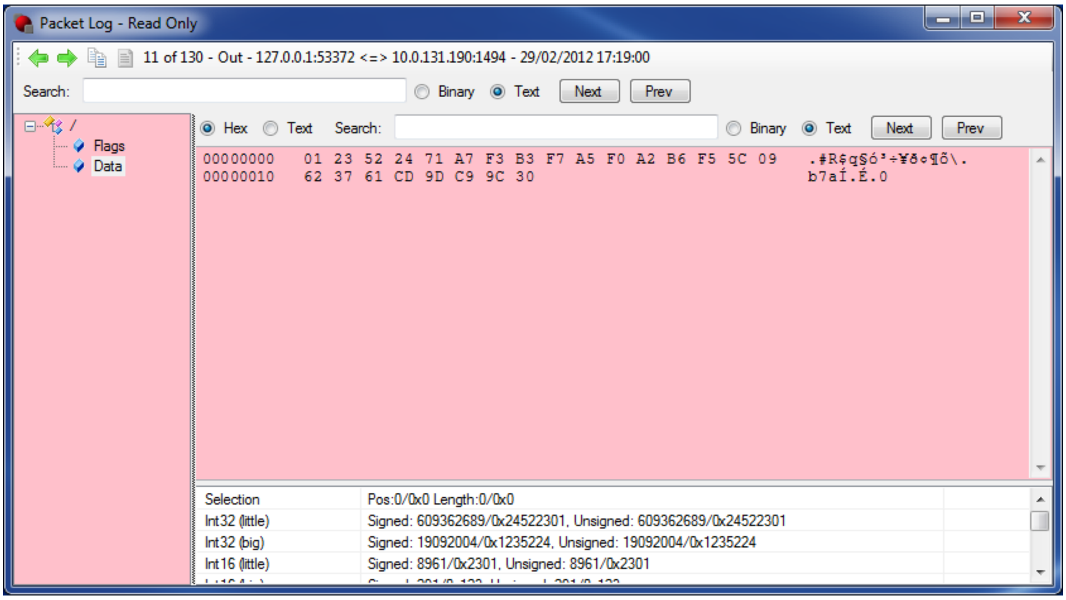


Figure 9 - Main Protocol Packet as a Tree

In order to allow for the packets to be converted back into a binary form each packet carries with it the information required to serialize to a stream even if the length of the data changes. This is especially important as it allows Canape to copy packets around, isolating them from the network connection they originally came from. With the packets in this form it is now possible to remove the encryption.

Removing the Encryption

The default encryption used by Citrix is effectively a basic XOR cipher with a 1 byte key, not the most secure of protocols. Deriving the actual algorithm is fairly trivial, but the simplest way is to decompile the Java client and directly extract the algorithm.

```
public EncryptProtocolDriver()
{
    super(false, g);
    h = false;
    i = false;
    l = (byte)(new Random()).nextInt();
    j = (byte)(l | 0x43);
    k = (byte)(l | 0x43);
}

private final void b(byte abyte0[], int i1, int j1)
{
    int k1 = (i1 + j1) - 1;
    byte byte0 = abyte0[k1];
    byte byte1 = l;
    for(int l1 = k1; l1 > i1; l1--)
        abyte0[l1] ^= abyte0[l1 - 1] ^ byte1;

    abyte0[i1] ^= j ^ byte1;
    j = byte0;
}
```

Figure 10 - Java Code for Encryption Algorithm

As the encryption is a property of the connection rather than the individual packets (as it uses the value of the previous encrypted byte to determine the value of the next one) this cannot just be applied to the packet and copied around like with the parsing of the framing. Instead it must be applied in the connection itself, with individual decrypt and encrypt nodes (as shown in Figure 11). The graph shown is actually containing in a sub-graph of the original state model (represented by a single node in the graph). This allows easy reuse of the discrete functionality and reduces complexity.

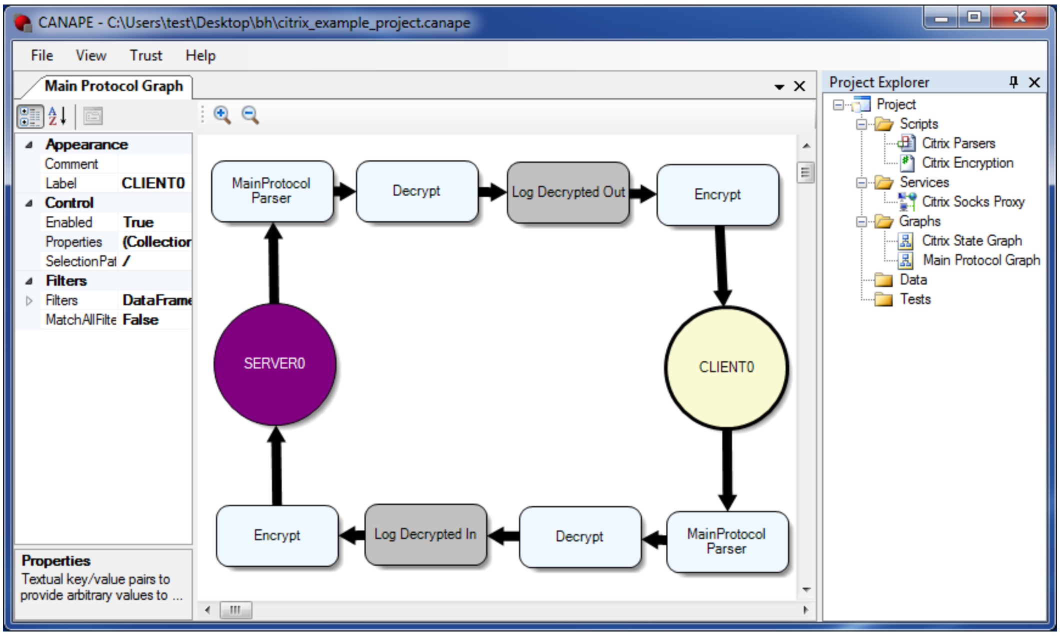


Figure 11 - Main Protocol Graph with Crypto

The encryption and decryption nodes are implemented in custom code, as it falls outside of basic parsing; this is however the only custom code required in the entire example project. For the full code see the example project supplied with this whitepaper.

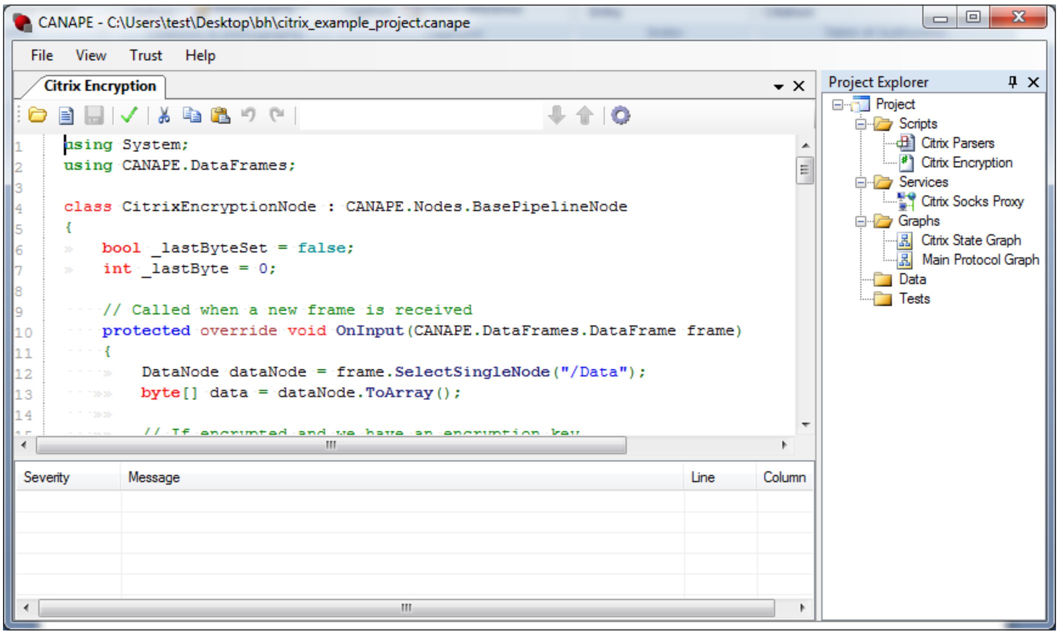


Figure 12 - Encryption Code

Disabling Compression

Now that the encryption has been removed it exposes that the underlying protocol is also compressed. The compression code is unfortunately not as simple as the basic encryption and it is also proprietary so we cannot easily repurpose existing code such as ZLib to decompress it. Also the Java Client has many hundreds of lines of obfuscated code making it difficult to extract. It is possible to disable it in the client through a registry modification but it would be preferable to be able to do it on the wire.

To find out how to do this the following registry key was set to disable compression and the packets compared.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Citrix\ICA
Client\Engine\Configuration\Advanced\Modules \TCP/IP\Compress = Off
```

This identified a single change in the initial negotiation packets, if compressed a specific set of bytes is set to 10 12, if compression is disabled they are set to 00 00.

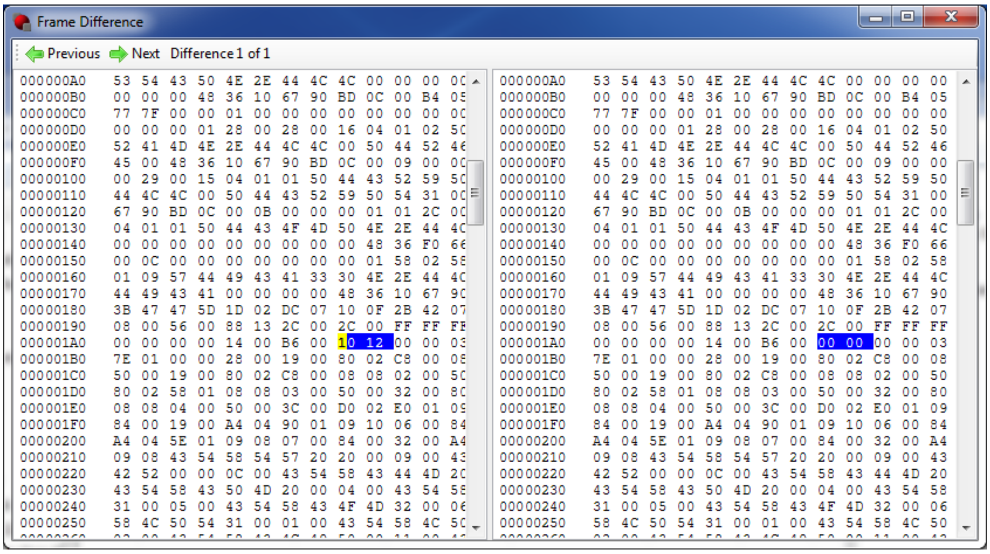


Figure 13 - Packet Differences between Compressed and Uncompressed

Using a built in node type in Canape it is possible to do arbitrary binary replacements to change the values to zeros before passing along to the server. Doing this allows the project to disable encryption without having to change the client's configuration any more than necessary.

The example project is now complete, it is possible to now add functionality to fuzz and modify packets as they traverse the network and find vulnerabilities. The next section describes one such issue which was previously identified and subsequently has been fixed by Citrix.

Citrix Vulnerability

This section of the whitepaper discusses the technique used to exploit the Citrix client vulnerability. The vulnerability itself is quite old, originally found in February 2008 but took the Vendor 2 years to fully fix the issue in all the affected clients[v]. Context found and exploited the vulnerability on Windows XP SP2, but the patch covered the following clients:

- Windows
- Linux (x86 and ARM)
- Solaris (Sparc and x86)
- Windows Mobile
- Mac

Attack Method

The attack works by enticing a victim to a malicious website which downloads an ICA file configured to connect to a fake ICA server. Standard web browsers with Citrix installed will automatically download an ICA file and pass it to the Citrix client which will then use the details in the file to connect to the ICA server.

The file is in the style of a simple INI file which contains the IP address of the target server. For the Proof of Concept that we developed the server was not a real Citrix server but an instance of Canape (in 2008 when the vulnerability was initially exploited, Canape was not available so custom code was used). The exploit is then sent to the ICA client and the machine exploited.

Memory Corruption Bug

The actual bug that was exploited was in the Citrix ThinWire virtual driver that is responsible for the graphics being displayed to the user. The bug was in an index overflow where a bounds check was not performed on data being received from the server. This issue was found by fuzzing the binary ICA protocol which resulted in the following crash:

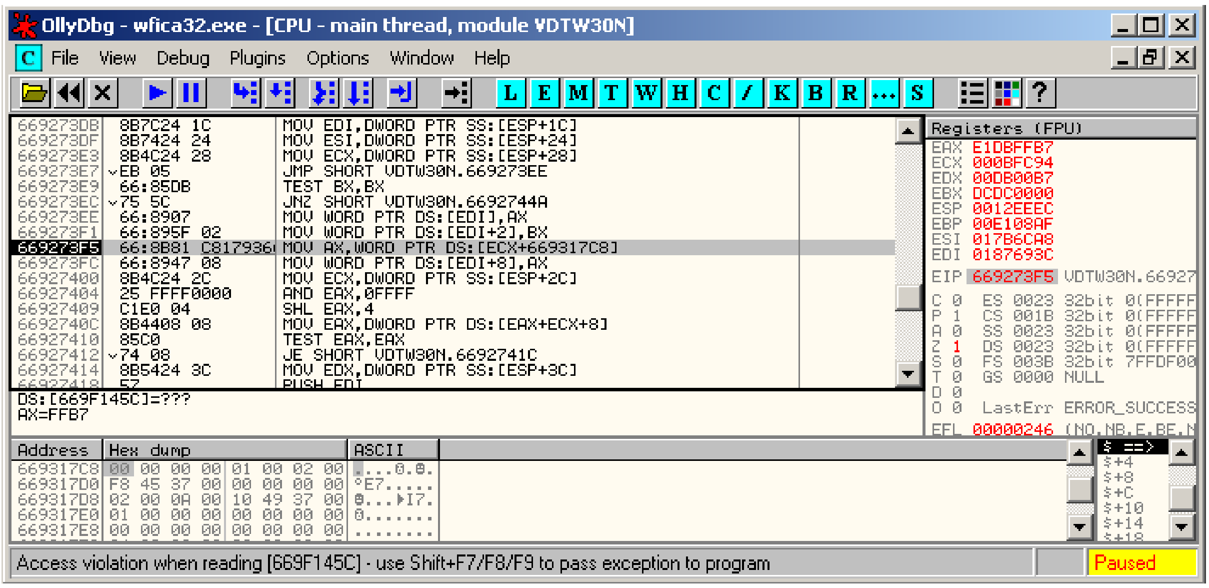
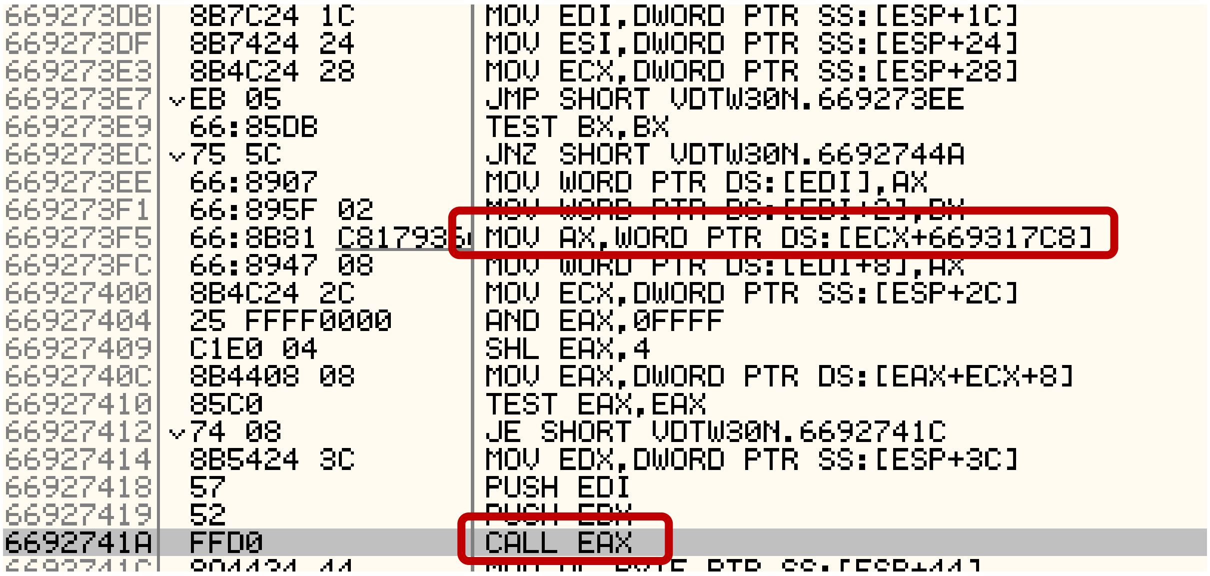


Figure 14 - Citrix crash during fuzzing.

This crash is in the VDTW30N module which is Virtual Driver Thin Wire responsible for the graphical updates. This crash is due to the value of ECX being out-of-bounds for the lookup at the fixed offset of 669317C8. The fuzz value used for this case was FFB7 which can be seen in the lower part of the EAX register. A read violation is an indication of a bug but is often not exploitable directly. Therefore further analysis of the crash and the subsequent code to determine code execution flow was employed. The following screenshot shows the surrounding code:



```
669273DB  8B7C24 1C        MOV EDI,DWORD PTR SS:[ESP+1C]
669273DF  8B7424 24        MOV ESI,DWORD PTR SS:[ESP+24]
669273E3  8B4C24 28        MOV ECX,DWORD PTR SS:[ESP+28]
669273E7 vEB 05            JMP SHORT VDTW30N.669273EE
669273E9  66:85DB          TEST BX,BX
669273EC v75 5C            JNZ SHORT VDTW30N.6692744A
669273EE  66:8907          MOV WORD PTR DS:[EDI],AX
669273F1  66:895F 02       MOV WORD PTR DS:[EDI+2],BX
669273F5  66:8B81 C81793   MOV AX,WORD PTR DS:[ECX+669317C8]
669273FC  66:8947 08       MOV WORD PTR DS:[EDI+8],AX
66927400  8B4C24 2C        MOV ECX,DWORD PTR SS:[ESP+2C]
66927404  25 FFFF0000      AND EAX,0FFFF
66927409  C1E0 04          SHL EAX,4
6692740C  8B4408 08        MOV EAX,DWORD PTR DS:[EAX+ECX+8]
66927410  85C0             TEST EAX,EAX
66927412 v74 08            JE SHORT VDTW30N.6692741C
66927414  8B5424 3C        MOV EDX,DWORD PTR SS:[ESP+3C]
66927418  57               PUSH EDI
66927419  52               PUSH EDX
6692741A  FFD0             CALL EAX
```

Figure 15 - Reverse Engineering the Crash

The first highlighted section shows the current crash location, the second shows a call to execute the memory at EAX. Further analysis shows that the value that we control influences the ultimate value of EAX at this point and thus where the code will be executed.

The complexity comes from determining how the value from the ICA packet is used to derive the value for EAX and therefore where the code is going to be executed. It was found to be easiest to brute force the value and examine the crashes that occur to find an input value which will result in the code running through to the call to EAX with a value which is in a memory area where we can influence. The brute forcing resulted in a value which caused the following crash:

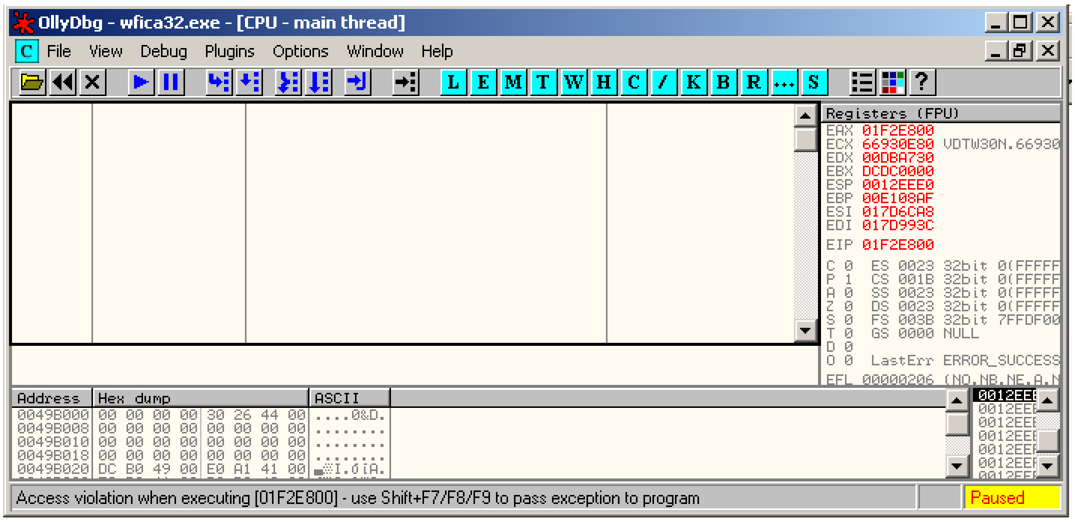


Figure 16 - Control EAX

As can be seen the program is trying to execute code at the address 01F2E800. There is currently no code at this address but by examining the memory layout we can work out if it is possible to heap spray up to that location:

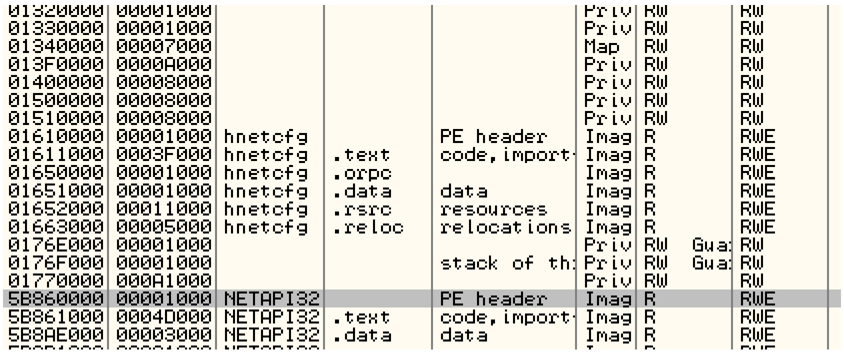


Figure 17 - Memory layout before heap spray

Currently the highest addressed heap block starts at offset 01770000 as can be seen in Figure 17, which is just below the location where the exploit will jump to. Therefore if we can get the application to allocate more heap memory with data that we control then we will call into the area where we have placed our data.

Heap Spraying

We used a standard heap spray technique to ensure that we have data at the location where the exploit will call. For ICA, we used a Thinwire Virtual Driver packet sent multiple times to fill the heap. The client was found to be allocating data for these packets but not releasing them. A second flaw in the Citrix client allowed us to cause large amounts of memory to be filled using only a smaller sized packet. This bug was in the way that the Citrix client would not check the length field within a packet and would copy the amount of data that was stated into memory. This data was copied out of a static buffer that was used to receive all ICA packets and therefore we could set a long length and it would copy that amount of data. The data that was not actually in the packet would be replaced with data from the previous packet.

So to heap spray the memory we send an initial large packet to prime the static packet buffer and then send thousands of small packets with a large inner length field to populate the heap. Figure 18 shows the memory layout that is the result of this flood.

```
01320000 00001000                                  Priv RW       RW
01330000 00001000                                  Priv RW       RW
01340000 00007000                                  Map  RW       RW
013C0000 00001000                                  Priv RW       RW
013E0000 0000A000                                  Priv RW       RW
013F0000 00008000                                  Priv RW       RW
014F0000 00008000                                  Priv RW       RW
01500000 00008000                                  Priv RW       RW
01600000 00001000 hnetcfg            PE header     Imag R        RWE
01601000 0003F000 hnetcfg  .text     code,import   Imag R        RWE
01640000 00001000 hnetcfg  .orpc                   Imag R        RWE
01641000 00001000 hnetcfg  .data     data          Imag R        RWE
01642000 00011000 hnetcfg  .rsrc     resources     Imag R        RWE
01653000 00005000 hnetcfg  .reloc    relocations   Imag R        RWE
0175E000 00001000                                  Priv RW  Gua: RW
0175F000 00001000                    stack of th:  Priv RW  Gua: RW
01760000 001FE000                                  Priv RW       RW
01D40000 003FE000                                  Priv RW       RW
02140000 00721000                                  Priv RW       RW
5B860000 00001000 NETAPI32           PE header     Imag R        RWE
5B861000 0004D000 NETAPI32 .text     code,import   Imag R        RWE
5B8AE000 00003000 NETAPI32 .data     data          Imag R        RWE
5B8B1000 00001000 NETAPI32 .rsrc     resources     Imag R        RWE
```

Figure 18 - Memory layout post heap spray

Thus offset 01F2E800 is now a valid address. However, the exact data at this location is not deterministic. It maybe a valid heap allocation from our heap spray or it might be in a block of zero bytes which are between allocations. Therefore we needed the exploit to execute through the zeroed block, into the data we control, through our NOP sled ultimately into the shellcode. A zero block of memory is disassembled in Figure 19.

```
01F2F1DA  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1DC  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1DE  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1E0  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1E2  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1E4  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1E6  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1E8  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1EA  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1EC  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1EE  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1F0  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1F2  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1F4  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1F6  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1F8  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1FA  0000   ADD BYTE PTR DS:[EAX],AL
01F2F1FC  0000   ADD BYTE PTR DS:[EAX],AL
```

Figure 19 - Zero memory NOP sled

This is a valid NOP instruction for this exploit as EAX will be pointing to a writable location. We know that we call the value of EAX and therefore it is guaranteed to be on the heap.

The next section of bytes that will be executed will be a heap header block. In Windows XP (which was used for the PoC) the header has the following structure:

Size	Prev Size	Cookie	Flags	Unused	Segment Index
0	2	4	5	6	7

8

Figure 20 - Heap Header Layout

The first two bytes are the size of the allocation (in 8 byte units); this is derived from the size of the packet and therefore is something we control. The other values are all fixed with the exception of the cookie value which is random. We therefore need this header block to be interpreted as instructions which causes no serious side effects. By examining the x86 instruction set a value was found that would ensure the dynamic value in the header block would be safely executed in both alignment situations. This value is 8100 as can be seen in Figure 21. The zeros are the unallocated data this is followed by the head size which has been set to 8100. This represents an ADD instruction to where EAX points to with the DWORD value in the following four bytes. Due to the fact that EAX is a valid pointer, this command will consume the random cookie value safely.

```
0000              ADD BYTE PTR DS:[EAX],AL
0000              ADD BYTE PTR DS:[EAX],AL
8100 B1016901     ADD DWORD PTR DS:[EAX],16901B1
0C 03             OR AL,3
90                NOP
90                NOP
```

Figure 21 - NOP heap header

The OR instruction is also safe because it has only a minor side effect on EAX. Therefore the heap flood packets must result in a memory allocation of 0x0408 bytes in length (which is 0x81 multiplied by 8). By doing this the exploit will execute from the heap through the zeros, over the heap header and into the shellcode. It was also necessary to place a few jumps in specific places in the packet to ensure it was a valid ThinWire packet but would still be executed.

Putting It All Together

This final section describes how Canape can be used to exploit the vulnerability described in the previous section using its built-in functionality. The tool supports the development of custom networking services and clients. This allows a final exploiting server to be developed entirely within Canape for demonstrating the vulnerability.

The Full Exploit

The steps for the exploit to work are as follows:

1. The victim visits a malicious site (and they have Citrix installed)
2. The site sends an ICA file to the client.
3. The ICA file instructs the client to connect to the malicious Citrix server.
4. The fake server then sends the hello and initial negotiation packets.
5. When the main stream is established a large packet is sent with the NOP sled and shellcode to prime the heap.
6. 3000 small packets are sent with a large length field to fill the heap.
7. Finally the exploit trigger packet is sent to cause the offset overflow that executes the shellcode.

Configuring the Replay Server

To effectively replay the traffic from server to client the packets first need to be placed into a separate packet log. This allows the built-in replay services to access the required data.

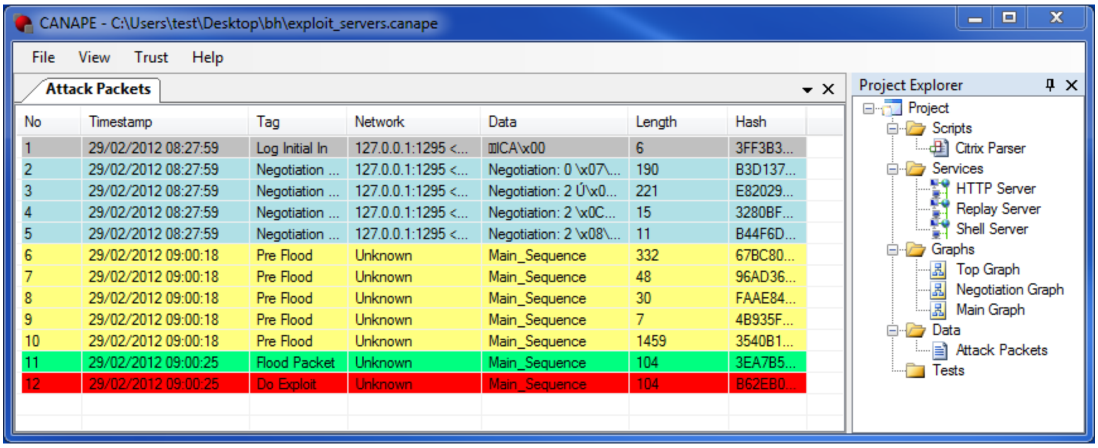


Figure 22 - Attack Packets

Each individual phase is marked with a special 'Tag' value which is used by the replay server to select the appropriate packet to send.

The replay server needs to be configured; this is done by creating a new network server and specifying the 'Full Replay Server' type. The configuration of this server contains a set of filters which match on specific packet data. When a match is made a 'Tag' is selected and the server sends back only those packets which match the tag.

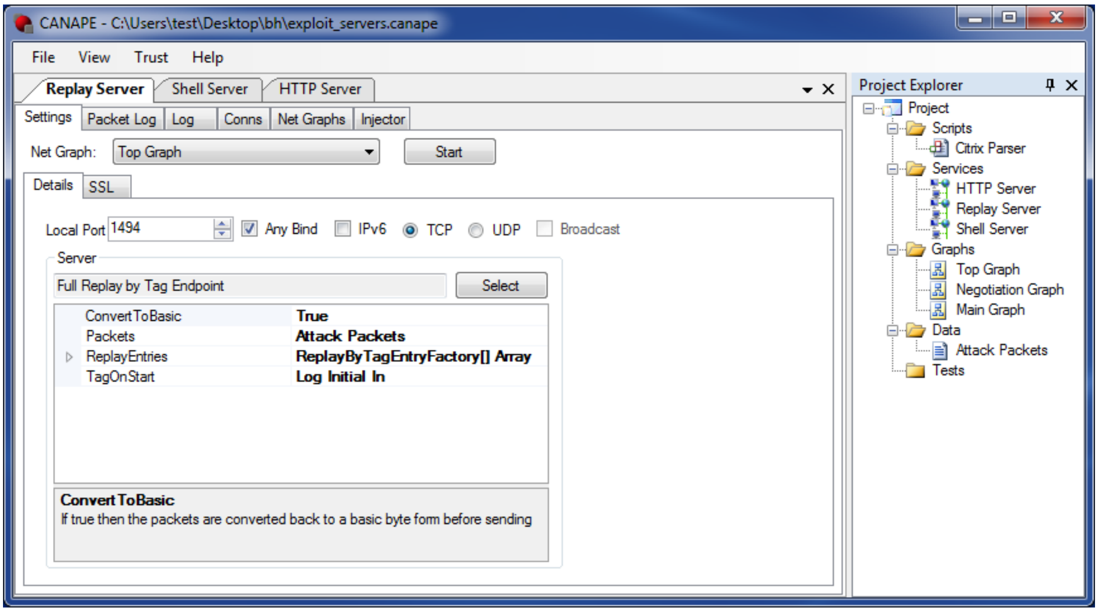


Figure 23 - Server Configuration

Setting Up HTTP and Remote Shell Server

The HTTP and Remote shell servers are configured in a similar way. For HTTP support Canape contains a very basic HTTP server which will send back a simple block of data to a HTTP request. For the Remote Shell a simple TCP server can be configured on port 4444 (which is specific to the shell code).

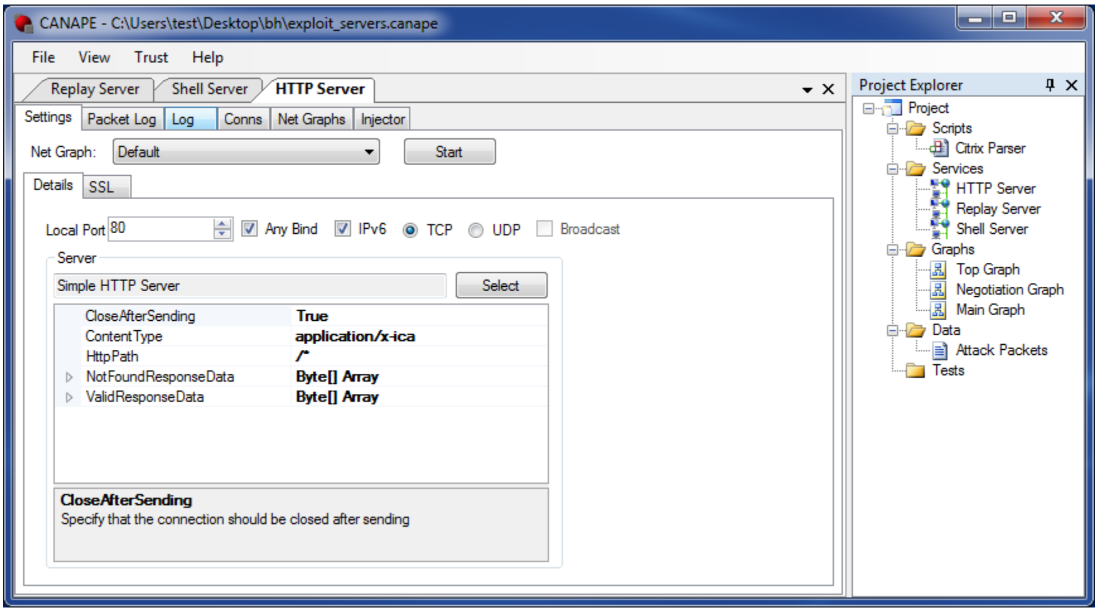


Figure 24 - HTTP Server Configuration

A web browser can now be used to retrieve the ICA file which sets the whole exploit process in motion.

The final packet sent to the ICA client is the one used to exploit the vulnerability; Figure 25 shows the exploit packet with the vulnerable value highlighted.

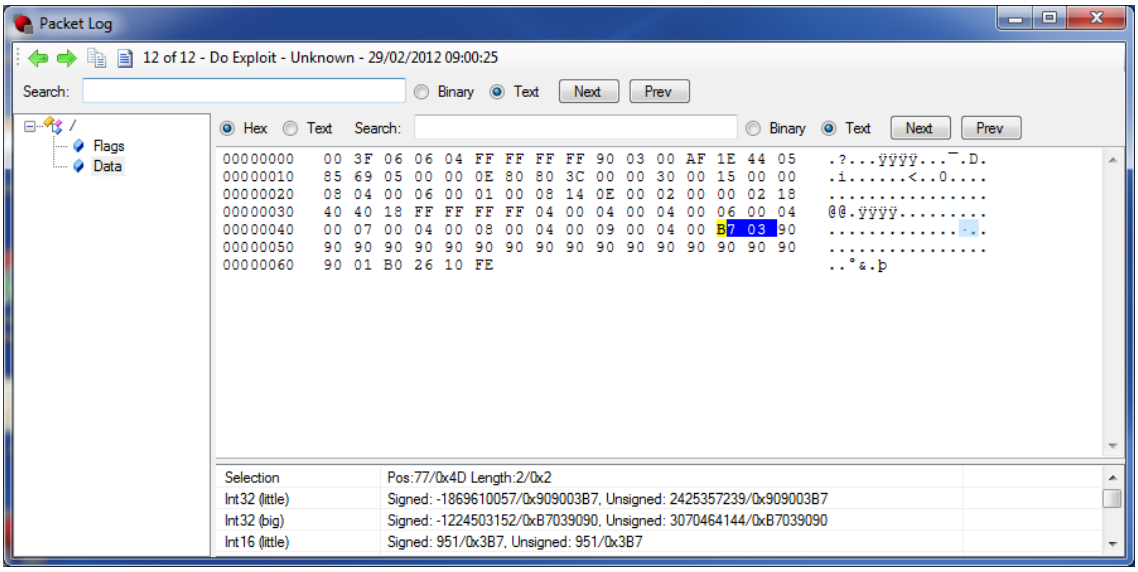


Figure 25 - The Exploit Packet

The reverse shell connection should now be available.

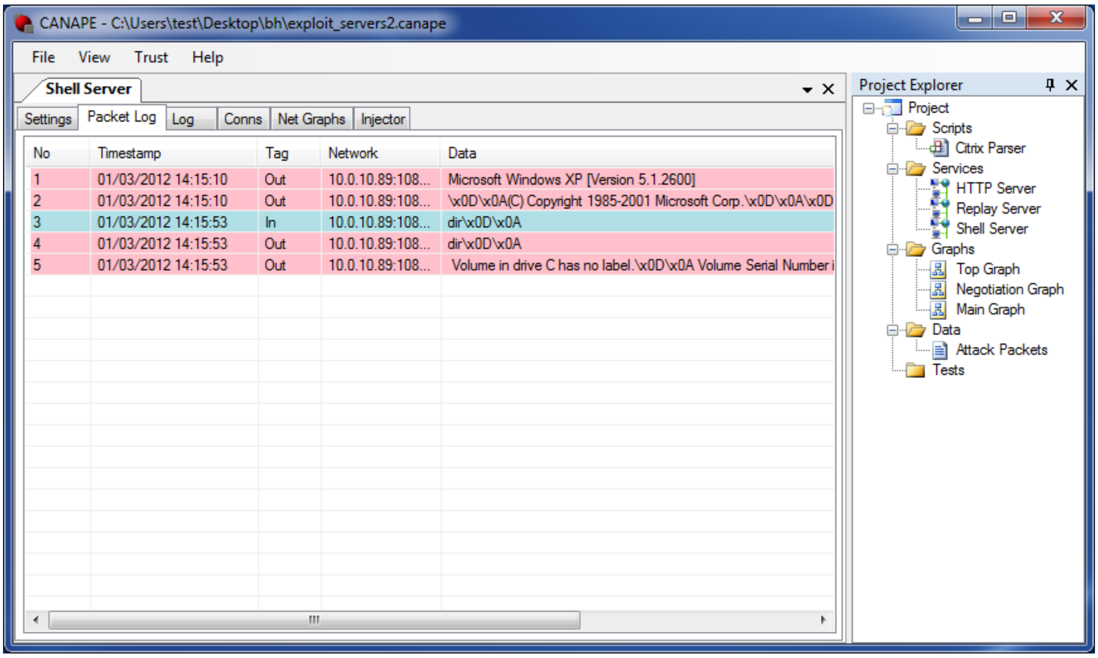


Figure 26 - Reverse Shell Connection

Conclusion

This whitepaper has demonstrated the process through which a bespoke binary protocol can be analysed and manipulated in Canape without a substantial amount of development effort. There is nothing particularly special in the use of Citrix ICA for this demonstration, the tool can equally be used to develop frameworks for other protocols, it is not even restricted to binary as text based protocols can be handled as well.

Further information on the usage of Canape as well as numerous tutorials is available on the project website, http://canape.contextis.com.

About Context

Context Information Security is an independent security consultancy specialising in both technical security and information assurance services.

The company was founded in 1998. Its client base has grown steadily over the years, thanks in large part to personal recommendations from existing clients who value us as business partners. We believe our success is based on the value our clients place on our product-agnostic, holistic approach; the way we work closely with them to develop a tailored service; and to the independence, integrity and technical skills of our consultants.

Context are ideally placed to work with clients worldwide with offices in the UK, Australia and Germany.

The company's client base now includes some of the most prestigious blue chip companies in the world, as well as government organisations.

The best security experts need to bring a broad portfolio of skills to the job, so Context has always sought to recruit staff with extensive business experience as well as technical expertise. Our aim is to provide effective and practical solutions, advice and support: when we report back to clients we always communicate our findings and recommendations in plain terms at a business level as well as in the form of an in-depth technical report.

context
INFORMATION SECURITY

Works Cited

---

[i] Context App Tool - http://cat.contextis.com
[ii] Fiddler - http://fiddler2.com/fiddler2/
[iii] Canape - http://canape.contextis.com
[iv] FreeCap - http://www.freecap.ru/eng/
[v] Citrix Security Advisory - http://support.citrix.com/article/CTX125975

Context Information Security Ltd

London (HQ)
4th Floor
30 Marsh Wall
London E14 9TP
United Kingdom

Cheltenham
Corinth House
117 Bath Road
Cheltenham GL53 7LS
United Kingdom

Düsseldorf
Adersstr. 28, 1.OG
D-40215 Düsseldorf
Germany

Melbourne
Level 9
440 Collins Street
Melbourne
Australia