

Inspecting and Attacking the Citrix ICA Protocol using Canape

James Forshaw and Michael Jordon whitepapers@contextis.com

Date: 16 March 2012

# Contents

Introduction	3
Canape Overview	4
Modelling the Protocol State	9
Parsing the Main Protocol	10
Removing the Encryption	12
Disabling Compression	14
Citrix Vulnerability	15
Attack Method	15
Memory Corruption Bug	15
Heap Spraying	17
The Full Exploit	20
Configuring the Replay Server	20
Setting Up HTTP and Remote Shell Server	21
Conclusion	23
About Context	24
Works Cited	25



## 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<sup>i</sup> and Fiddler<sup>ii</sup>, 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 <a href="http://canape.contextis.com">http://canape.contextis.com</a>

## **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<sup>iv</sup>) 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:

i≯l	e	
ויח		
	Г	71
		· .

Citrix	Socks Proxy						<del>~</del> ×	Project Explorer	ņ
ettings	Packet Log Log	Conns	Net Graphs	Injector				Project —────────────────────────────────────	
No	Timestamp		Tag	Network	Data	Length	Hash 🔺	E Dervices	
1	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	DICA\x00	6	3FF3B ≡	Citrix Socks Prox	1
2	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	DICA\x00	6	3FF3B	Data	
3	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	\x00a\x00\x07\x00	243	99310.	Tests	
4	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x01~\x00x\~18	1456	76F01		
5	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x00H6òf}\x00\x00f	740	A35D6		
6	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	\x02\x1F\x01\x1F\	331	DB9F5		
7	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x04\x00\x00	3	740EA		
8	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	\x02\x00\x02\x00	4	9EC7C		
9	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x03\x00\x04\x00,	5	95159		
10	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	\x12\x00\x01C+\x0	20	B2B25		
11	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x18\x00\x01cjdlçE	26	9E5BE		
12	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	\x05\x00\x01\yV	7	4FF24		
13	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	\x16A\x01ôÂêÆjÇ8	280	454C2		
14	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	E\x01 <sup>¶</sup> mMb&c+>3	1418	2E572		
15	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	A@\x01Ú\x06p:ÔÅ	67	6C3CF		
16	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	-\x00\x01Èäóù\x16	47	B4E0C		
17	29/02/2012 16:0	)1:16	Out	127.0.0.1:52899	8@\x01c3: 3%Z%os§	58	08B75		
18	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	-E\x01MGalg¶,]vB	1454	D681F		
19	29/02/2012 16:0	)1:16	In	127.0.0.1:52899	iB\x01ÕãnY\x0FÉÑ	619	BB51F		
20	29/02/2012 16:0	1.10	In	127.0.0.1:52899	\x00\x01/ÑY³ÝIÔ6c	146	722E6		

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.

👇 Packet Log - Re	ad Only	_ <b>-</b> ×
🔶 🔶 🖣 🗎	1 of 2363 - In - 127.0.0.1:50257 <=> 10.0.131.190:1494 - 29/02/2012 13:03:53	
🖲 Hex 🔘 Text	Search: O Binary O Text Next Prev	
0000000 <b>7</b> F	7F 49 43 41 00 .ICA.	
		-
Selection	Pos:0/0x0 Length:0/0x0	
Int32 (little)	Signed: 1128890239/0x43497F7F, Unsigned: 1128890239/0x43497F7F	E
Int32 (big)	Signed: 2139048259/0x7F7F4943, Unsigned: 2139048259/0x7F7F4943	
Int 16 (little)	Signed: 32639/0x7F7F, Unsigned: 32639/0x7F7F	
Int16 (big)	Signed: 32639/0x7F7F, Unsigned: 32639/0x7F7F	

#### 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.

### White paper/Canape: Bytes your Bits

) Hex () Te			213 -	Out	- 127													
	ext	C				1.0.0.1	L:531	11 <	=>1	10.0.:	131.1	90:1	494 -	29/0	)2/2(	012 16	:15:43	
		Sear	rch:												$\bigcirc$	Binary	Text     Next     Prev	
	01		~~		00		00	01	07	77	66	69	63	61	33		."."wfica32	
			78	65	00	00	00	00	00	00	00	00	00	00	00		.exeH	
	36	10	67	90	BD	0C	00	24	00	00	00	09	08	00	00	01	6.g.¥\$	
0000030	6C	00	00	00	00	00	00	5F	00	00	00	00	00	00	00	00	1	
00000040	00	00	00	00	00	SE .	CB	89	00	51	00	49	00	00	00	00 1 F	ĒI	
	00	00	20	22	47	76	60	60	60	25	72	31	25	14	11	11	,zJvhk`~sr>	
	18	0.7	2C 4B	OF	18	10	15	50	60	15	13	52	10	49	50	6D		
	48	67	412	06	29	29	20	25	27	3F	OF	63	5B	72	4D		HgJ.)), %'?.c[rMu	
	34	08	27	4B	6E	41	6C	20	0F	0 F	AO	03	01	19	75	28	4.'KnAlu(	
	74	36	6A	36	74	20	73	2E	73	2 F	43	01	3Ĉ	00	30		t6j6t-s.s/C.<.<.	
000000B0	17	0.5	01	02	54	44	57	53	54	43	50	4E	2E	44	4C	4C	TDWSTCPN.DLL	
00000000	00	00	00	00	00	00	00	00	00	00	48	36	10	67	90	BD	H6.g.¥	
00000D0	0C	00	Β4	05	02	00	CF	77	7 F	00	00	01	00	00	00	00	´Ïw	
00000E0	00	00	00	00	00	00	00	00	00	00	01	28	00	28	00	16		
0000010	04	01	02	50	44	52	46	52	41	4D	4E	2E	44	4C	4C	00	PDRFRAMN.DLL.	
00000100	50	44	52	46	52	41	4 D	45	00	48	36	10	67	90	BD	0C	PDRFRAME.H6.g.∺.	
00000110	00	09	00	00	00	01	29	00	29	00	15	04	01	01	50	44	).)PD	
00000120	43	52	59	50	54	4 E	2E	44	4C	4C	00	50	44	43	52	59	CRYPTN.DLL.PDCRY	
00000130	50	54	31	00	48	36	10	67	90	BD	00	00	08	00	00	00	PT1.H6.g.%	
Selection			Pos	:0/0x	0 Ler	ngth:0	)/0x0	)										
Int32 (little)			Sign	ed: -	1476	35199	99/06	xA80	0A80	1, Ur	nsigne	ed: 2	8186	1529	7/0x	A800A	801	[
Int32 (bia)			-			7432/					-							
nt 16 (little)			_			7/0xA				_								
nt16 (iitie)			_			1/1/000			-			M-101						

#### 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 littleendian length field, followed by a 4 bit set of flags. This is followed by the number of bytes indicated in the length field.

🕐 Packet Log - Read O	niy	x
<ul> <li></li></ul>	213 - In - 127.0.0.1:53111 <=> 10.0.131.190:1494 - 29/02/2012 16:15:43 ch:	*
00000030 B7 24 00000040 4C F9 00000050 1D 86 00000050 69 64 00000070 58 42 00000020 B5 7D 00000080 B5 7D 0000080 05 E7 0000080 19 4E	3D 32 /5 /1 D 2E E9 5B 9A 27 28 C9 36 8B 3D       -,i,i,i,u,i,u,i,u,i,u,i,u,i,u,i,u,i,u,i,	
000000D0 55 C6 000000E0 3D 39 000000F0 02 78	92 17 35 0A 10 98 F9 89 93 5E C6 D4 4F 5A U£5ù^£ÔOZ A2 30 21 0B DA 87 45 1D 44 8D D0 52 A5 4F =9s0!.Ú.E.D.DR¥O 13 AF 20 97 95 BE 95 C2 B0 A0 ED 3B 82 23 .x. <sup>-</sup> 4.Ű 1; 21 CA CD 32 5D 43 6D 9B D1 1A ED 26 C5 3E µ.!ÉÍ2]Cm.Ñ.ísÅ≻	Ŧ
Selection	Pos:0/0x0 Length:0/0x0	_
Int32 (little) Int32 (big)	Signed: 1812021521/0x6C014111, Unsigned: 1812021521/0x6C014111 Signed: 289472876/0x1141016C, Unsigned: 289472876/0x1141016C	=
Int16 (little)	Signed: 16657/0x4111, Unsigned: 16657/0x4111	
Int16 (big)	Signed: 4417/0x1141, Unsigned: 4417/0x1141	-

#### 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.

CANAPE - C:\Users\test\Desktop	p\bh\citrix_ex	ample_project.canape				x
File View Trust Help						
Citrix Parsers				<b>→</b> ×	Project Explorer	ųΧ
Project Enums Sequences Parsers Citrix_Frame Citrix_Frame_Parser	Name Length Rags Data Data Data Data A Behav Default A Membel Descrip Uuid A Misc Membel Default En The endian	iour Endian ation tion 'Count	Size 2 1 1 1  BigEndian 39789714-6783-401f-a23 3	9-6472ae63d23e	Project	ryption ks Proxy e Graph

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.

🕐 Packet Log - Read Onl	у		
🧯 🦛 🚔 📄 11 of 1	30 - Out - 127.0.0.1:53372	<=> 10.0.131.190:1494 - 29/02/2012 17:19:00	
Search:		Binary      Text     Next     Prev	
	ío Hex ⊙ Text Sea	arch: O Binary   Text  Prev	
Data		52 24 71 A7 F3 B3 F7 A5 F0 A2 B6 F5 5C 09 .#Req§ó³+¥ðe¶õ∖. 61 CD 9D C9 9C 30 b7aĨ.Ĕ.0	4
	Selection	Pos:0/0x0 Length:0/0x0	-
	Int32 (little)	Signed: 609362689/0x24522301, Unsigned: 609362689/0x24522301	
	Int32 (big)	Signed: 19092004/0x1235224, Unsigned: 19092004/0x1235224	
	Int16 (little)	Signed: 8961/0x2301, Unsigned: 8961/0x2301	-

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)(1 | 0x43);
    k = (byte)(1 | 0x43);
3
private final void b(byte abyte0[], int i1, int j1)
£
    int k1 = (i1 + j1) - 1;
    byte byte0 = abyte0[k1];
    byte byte1 = 1;
    for(int l1 = k1; l1 > i1; l1--)
        abyte0[11] ^= abyte0[11 - 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 subgraph 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.

CANAPE - C:\Users\test\Desktop\bh\citrix_example_project.canape		-	
File View Trust Help			
Citrix Encryption		• ×	Project Explorer 4 ×
Gitrix Encryption         i       ising System;         2       using CANAPE.DataFrames;         3       class CitrixEncryptionNode : CANAPE.Nodes.Base;         5       {         6       bool_lastByteSet = false;         7       int_lastByte = 0;         8	caFrames.DataFrame frame) eNode("/Data");	× ×	Project Explorer 4 ×
< III		•	

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.

膏 Frame Diffe	renc	e																									>	3
🔶 🔶 Previous	<b> </b>	Vext	Diff	feren	ce 1	of 1																						
000000A0	53	54	43	50	4E	2E	44	4C	4C	00	00	00	A 00	000000A0	53	54	43	50	4 E	2E	44	4C	4C	00	00	00	00	
000000B0	00	00	00	48	36	10	67	90	BD	0C	00	Β4	0.5	000000B0	00	00	00	48	36	10	67	90	BD	0C	00	Β4	05	
000000C0	77	7 F	00	00	01	00	00	00	00	00	00	00	00	000000C0	77	7 F	00	00	01	00	00	00	00	00	00	00	00	
000000D0	00	00	00	01	28	00	28	00	16	04	01	02	50	000000D0	00	00	00	01	28	00	28	00	16	04	01	02	50	
000000E0	52	41	4 D	4E	2E	44	4C	4C	00	50	44	52	4€	000000E0	52	41	4 D	4 E	2E	44	4C	4C	00	50	44	52	46	
000000F0	45	00	48	36	10	67	90	BD	0C	00	09	00	00	000000F0	45	00	48	36	10	67	90	BD	0C	00	09	00	00	
00000100	00	29	00	15	04	01	01	50	44	43	52	59	50	00000100	00	29	00	15	04	01	01	50	44	43	52	59	50	
00000110	44	4C	4C	00	50	44	43	52	59	50	54	31	0 C 🗏 🛛	00000110	44	4C	4C	00	50	44	43	52	59	50	54	31	00	E
00000120	67	90	BD	0C	00	0B	00	00	00	01	01	2C	0 C	00000120	67	90	BD	0C	00	0B	00	00	00	01	01	2C	00	
00000130	04	01	01	50	44	43	4 F	4 D	50	4E	2E	44	4C	00000130	04	01	01	50	44	43	4 F	4 D	50	4E	2E	44	4C	
00000140	00	00	00	00	00	00	00	00	00	48	36	FO	66	00000140	00	00	00	00	00	00	00	00	00	48	36	FO	66	
00000150	00	0C	00	00	00	00	00	00	00	01	58	02	58	00000150	00	0C	00	00	00	00	00	00	00	01	58	02	58	
00000160	01	09	57	44	49	43	41	33	30	4E	2E	44	4C	00000160	01	09	57	44	49	43	41	33	30	4E	2E	44	4C	
00000170	44	49	43	41	00	00	00	00	48	36	10	67	90	00000170	44	49	43	41	00	00	00	00	48	36	10	67	90	
00000180	3B	47	47	5D	1D	02	DC	07	10	0 F	2B	42	07	00000180	3B	47	47	5D	1D	02	DC	07	10	0 F	2B	42	07	
00000190	08	00	56	00	88	13	2C	00	2C	00	FF	FF	FF	00000190	08	00	56	00	88	13	2C	00	2C	00	FF	FF	FF	
000001A0	00	00	00	00	14	00	B6	00	10	12	00	00	03	000001A0	00	00	00	00	14	00	B6	00	00	00	00	00	03	
000001B0	7E	01	00	00	28	00	19	00	80	02	C8	00	30	000001B0	7E	01	00	00	28	00	19	00	80	02	C8	00	08	
000001C0	50	00	19	00	80	02	C8	00	08	08	02	00	50	000001C0	50	00	19	00	80	02	C8	00	08	08	02	00	50	
000001D0	80	02	58	01	08	08	03	00	50	00	32	00	80	000001D0	80	02	58	01	08	08	03	00	50	00	32	00	80	
000001E0	08	08	04	00	50	00	3C	00	DO	02	Ε0	01	20	000001E0	08	08	04	00	50	00	3C	00	D0	02	ΕO		09	
000001F0	84	00	19	00	Α4	04	90	01	09	10	06	00	84	000001F0	84	00	19	00	Α4	04	90	01	09	10	06	00	84	
00000200	A4	04	5E	01	09	08	07	00	84	00	32	00	A4	00000200	A4	04	5E	01	09	08	07	00	84	00	32	00	A4	
00000210	09	08	43	54	58	54	57	20	20	00	09	00	43	00000210	09	08	43	54	58	54	57	20	20	00	09	00	43	
00000220	42	52	00	00	0C	00	43	54	58	43	44	4 D	20	00000220	42	52	00	00	0C	00	43	54	58	43	44	4 D	20	
00000230	43	54	58	43	50	4 D	20	00	04	00	43	54	58	00000230	43	54	58	43	50	4 D	20	00	04	00	43	54	58	
00000240	31	00	05	00	43	54	58	43	4 F	4D	32	00	0€	00000240	31	00	05	00	43	54	58	43	4 F	4D	32	00	06	
00000250	58	4C	50	54	31	00	01	00	43	54	58	4C	5C 🖵	00000250	58	4C	50	54	31	00	01	00	43	54	58	4C	50	-
00000000	0.0	00	10	<b>F</b> 4	5.0	10	40	4.0		00	4.4	00	40	00000000	0.0	00	10	<b>F</b> 4	5.0	10	40	4.0		00	4.4	0.0	4.0	

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<sup>v</sup>. 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:

0	
CollyDbg - wfica32.exe - [CPU - main thread, module VDTW30N]	
C File View Debug Plugins Options Window Help	
6692730B         887C24         1C         MOU EDI, DUNORD PTR SS: LESP+1C1           6692730B         887C24         24         MOU ESI, DUNORD PTR SS: LESP+241           6692730F         887C24         28         MOU ECX, DWORD PTR SS: LESP+281           669273E7         VEB 05         JHP SHORT VUTW30H.669273EE           669273E7         VEB 05         JHP SHORT VUTW30H.669273EE           669273E2         VES 05         JHZ SHORT VUTW30H.6692744A           669273E2         66:85DB         TEST BX, BX           669273E2         66:895F 02         MOU WORD PTR DS: LEDI+21, BX           669273E3         66:895F 02         MOU WORD PTR DS: LEEDI+21, BX           669273E4         66:8981 C31:7681         000 WORD PTR DS: LEEDI+31, AX           669273E4         66:8981 C31:7681         000 WORD PTR DS: LEEDI+31, AX           669274E9         66:8947 08         MOU WORD PTR DS: LEEI+83, AX           66927404         25 FFFF0000         AND EAX, OWORD PTR SS: LESP+2C1           66927404         25 FFFF0000         AND EAX, OWORD PTR DS: LEAX+ECX+81           66927404         25 FFFF0000         AND EAX, OWORD PTR DS: LEAX+ECX+81           66927405         884408 08         MOU WORD PTR DS: LEAX+ECX+81           66927410         885424 30         MOU EOX, DWO	Registers         (FPU)           EAX         E1DBFFB7           ECX         000B008F094           EDX         00DB008F094           EDX         00DB0008F094           EDX         00DB0008F094           EDX         00DB0008F094           EDX         DCDC00000           ESP         0012EEEC           EBP         0012EEEC           EII         01786C08           EDI         0187693C           EIP         669273F5           VDTW30N.66927         C           C         0           ES         0018           SUIT         0(FFFFF           A         0           S         0023           S         0023           EDI         0           S         0023           S         0023           S         0023           D         0
DS:[669F145C]=??? AX=FFB7	0 0 LastErr ERROR_SUCCESS
Address Hex dump ASCII	EFL 00000246 (NO.NB.E.BE.N
669317C8       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00       00	\$+4 \$+2 \$+10 \$+10 \$+14 \$+14 \$+18
Access violation when reading [669F145C] - use Shift+F7/F8/F9 to pass exception to program	Paused

Figure 14 - Citrix crash during fuzzing.

### White paper/Canape: Bytes your Bits

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:



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:

🔆 OllyDbg - wfic	ca32.exe - [Cl	PU - main threa	d]						_ 🗆 ×
C File View D	Debug Plugins	Options Windo	w Help						_ 8 ×
	· II - 4 +	년 11 - 기	→ L E	MTW	IC/KE	3 R S		?	
							ECX 669 EDX 000 EBX DC0 ESP 001 EBP 000 ESI 017	F2E800	U) VDTW30N.66930
						•	EIP 01F C 0 ES P 1 CS A 0 SS Z 0 DS S 0 FS T 0 GS	5 001B 5 0023 5 0023 5 003B	32bit 0(FFFF 32bit 0(FFFF 32bit 0(FFFF 32bit 0(FFFF 32bit 7FFDF00 NULL
									ERROR_SUCCESS
0049801800 00	00 00 30 26 00 00 00 00 00 00 00 00 00 00 00 00	ASCII           44         00           00         00           00         00           00         00           00         00           00         00           41         00           41         00							▲ 0012EE
Access violation w	vhen executing [	(01F2E800) - use 9	hift+F7/F8/F9	to pass excepti	on to program				Paused

#### 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:

01320000 00001000 01340000 00001000 013F0000 00008000 0140000 00008000 01500000 00008000 01510000 00008000 01510000 00001000 hnetc 0161000 0001000 hnetc 01650000 0001000 hnetc 01652000 0001000 hnetc 01652000 0001000 hnetc 01652000 0001000 hnetc 0176F000 00001000 hnetc 0176F000 00001000 hnetc 0176F000 00001000 hnetc	g .text g .orpc g .data g .rsrc g .reloc	PE header code, import- data resources relocations stack of th: PE header	Priv RW Priv RW Priv RW Priv RW Priv RW Priv RW Imag R Imag R Imag R Imag R Imag R Imag R Priv RW Priv RW Priv RW Priv RW	RW RW RW RW RW RW RWE RWE RWE RWE RWE RW
58861000 0004D000 NETAP 588AE000 00003000 NETAP	32 .text	code,import	Imag R Imag R	RWE

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 0134000 00001000 0134000 00001000 013E0000 00001000 013E0000 00008000 014F0000 00008000 0150000 00008000 0160000 00001000 01640000 00001000 01642000 0001000 01642000 0001000 01653000 00005000 0175E000 00001000 0175E000 00001000 0175E000 00001000	hnetofg hnetofg hnetofg hnetofg	.text .orpc .data .rsrc .reloc	PE header code,import data resources relocations stack of th:	Imag Imag Imag Imag Priv	reservantes Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Secondration Second	Gua: Gua:		
01D40000 003FE000				Priv	RW		RW	
02140000 00721000 58860000 00001000 58861000 0004D000 588AE000 00003000 588AE000 00003000 588B1000 00001000	NETAPI32 NETAPI32	.text .data .rsrc	PE header code,import data resources	Priv Imag Imag Imag Imag	R₩ R R R		RW RWE RWE RWE 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	
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	ดัดดัด	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	ดัดดัด	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	2 4	1 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

### White paper/Canape: Bytes your Bits

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.

File	View Trust Help					• X	Project Explorer 4
No	Timestamp	Tag	Network	Data	Length	Hash	Project
1	29/02/2012 08:27:59	Log Initial In	127.0.0.1:1295 <	DICA\x00	6	3FF3B3	Citrix Parser
2	29/02/2012 08:27:59	Negotiation	127.0.0.1:1295 <	Negotiation: 0 \x07\	190	B3D137	📄 🗁 Services
3	29/02/2012 08:27:59	Negotiation	127.0.0.1:1295 <	Negotiation: 2 Ú\x0	221	E82029	HTTP Server
4	29/02/2012 08:27:59	Negotiation	127.0.0.1:1295 <	Negotiation: 2 \x0C	15	3280BF	
5	29/02/2012 08:27:59	Negotiation	127.0.0.1:1295 <	Negotiation: 2 \x08\	11	B44F6D	Shell Server
6	29/02/2012 09:00:18	Pre Flood	Unknown	Main_Sequence	332	67BC80	Graphs
7	29/02/2012 09:00:18	Pre Flood	Unknown	Main_Sequence	48	96AD36	Top Graph
8	29/02/2012 09:00:18	Pre Flood	Unknown	Main_Sequence	30	FAAE84	Negotiation Graph
9	29/02/2012 09:00:18	Pre Flood	Unknown	Main_Sequence	7	4B935F	Main Graph
10	29/02/2012 09:00:18	Pre Flood	Unknown	Main_Sequence	1459	3540B1	Data
11	29/02/2012 09:00:25	Flood Packet	Unknown	Main_Sequence	104	3EA7B5	Tests
12	29/02/2012 09:00:25	Do Exploit	Unknown	Main_Sequence	104	B62EB0	

#### 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.

CANAPE - C:\Users\test\Desktop\bh\exploit_servers.canape	
File View Trust Help	
Replay Server         Shell Server         HTTP Server         X	Project Explorer 4 X
Settings Packet Log Log Conns Net Graphs Injector	Scripts
Net Graph: Top Graph   Start	Citrix Parser
Details SSL	HTTP Server
Local Port 1494 🚔 🗹 Any Bind 🔲 IPv6 💿 TCP 💿 UDP 🗌 Broadcast	Replay Server
Server	⊡⁄⊒ Graphs /3 Top Graph
Full Replay by Tag Endpoint Select	
Convert To Basic True	
Packets Attack Packets Packets ReplayEntryFactory[] Array	Attack Packets
TagOnStart Log Initial In	Tests
Convert To Basic	
If true then the packets are converted back to a basic byte form before sending	

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).

CANAPE - C:\Users\test\Desktop\bl	\exploit_servers.canape								
File View Trust Help									
Replay Server Shell Server	HTTP Server	<b>-</b> ×	Project Explorer 4 ×						
Settings Packet Log Log Conns									
Net Graph: Default Details SSL Local Port 80	▼ Start	icast	Scripts						
Server			Graphs						
Simple HTTP Server	Select		····몶 Top Graph ····몶 Negotiation Graph						
CloseAfterSending Content Type HttpPath NotFoundResponseData ValidResponseData CloseAfterSending Specify that the connection should b	True application/x-ica /* Byte[] Array Byte[] Array Byte[] Array		Main Graph						

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.

Gearch:			🔘 Bina	iry 🤇	Text	N	ext	F	rev	]			
	Hex Text	Search									08	inary	Text     Next     Prev
Data	00000010 85 00000020 08 00000030 40 00000040 00 00000050 90		00 00 06 00 FF FF 0 04 00	0E 01 FF 08 90	FF 0 00 0	0 3C 8 14 4 00	00 0E 04 09	03 00 00 00 00 90	30 ( 02 ( 04 ( 04 (	AF 1E 00 15 00 00 00 06 00 <mark>B7</mark> 90 90	00 02 00 03	00 18 04 90	. 2 ÿÿÿ D. .i< . 0 @@. ÿÿÿ 
	Selection		os:77/0x4[										
	Int32 (little) Int32 (big) Int16 (little)	Si	gned: -186 gned: -122 gned: 951.	45031	52/0xB	70390	90, Ur	nsigne					

#### Figure 25 - The Exploit Packet

The reverse shell connection should now be available.

CANA	PE - C:\Users\test\Deskto	o\bh\exp	loit_servers2.canap	e	
File	View Trust Help				
Shel	Server			<b>▼</b> X	Project Explorer 🛛 📮 🗙
Settings	Packet Log Log Conr	ns Net G	àraphs Injector		□····⑦ Project □···/② Scripts
No	Timestamp	Tag	Network	Data	Citrix Parser
1	01/03/2012 14:15:10	Out		Microsoft Windows XP [Version 5.1.2600]	HTTP Server
2	01/03/2012 14:15:10	Out	10.0.10.89:108		Replay Server
3 4 5	01/03/2012 14:15:53	In	10.0.10.89:108		Shell Server
4	01/03/2012 14:15:53	Out	10.0.10.89:108		🖨 🦾 Graphs
5	01/03/2012 14:15:53	Out	10.0.10.89:108	Volume in drive C has no label.\x0D\x0A Volume Serial Number i	Top Graph
					Negotiation Graph
					Attack Packets
					Tests
•		1		4	

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, <u>http://canape.contextis.com</u>.

# 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.



# **Works Cited**

- <sup>i</sup> Context App Tool <u>http://cat.contextis.com</u>
- " Fiddler http://fiddler2.com/fiddler2/
- Canape <u>http://canape.contextis.com</u>
- iv FreeCap <u>http://www.freecap.ru/eng/</u>
- 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, 1.0G D-40215 Düsseldorf Germany

#### Melbourne

Level 9 440 Collins Street Melbourne Australia