FEATURE

WHAT ARE BROWSER EXPLOIT KITS UP TO? A LOOK INTO SWEET ORANGE AND PROPACK

Aditya K. Sood, Richard J. Enbody
Michigan State University, USA

Rohit Bansal
Independent Security Researcher, USA

At the VB2011 conference, our team discussed the techniques used by the Blackhole and Phoenix browser exploit packs (BEPs) [1] to spread malware. Blackhole has become a major player in the world of BEPs, but it is not the only one in demand. Sweet Orange and ProPack have recently entered the market, and both are gaining popularity. A simple traffic analysis of Sweet Orange can be found in [2]. In an earlier study [3] we discussed the details of the exploit distribution mechanism in BEPs. In this paper, we look at advancements in the design of BEPs, specifically Sweet Orange (SO) and ProPack.

SWEET ORANGE

iframe cryptor service

Today's BEPs provide automated iframe obfuscating services for use in web injections. The iframes are injected into high-traffic-volume websites and force the users of the websites to visit end points that serve exploits carrying malware. The SO BEP framework includes an iframe cryptor service for obfuscating iframes. This extends the capability of SO to obfuscate and inject the iframe at the same time, meaning that the attacker does not have to buy obfuscation services from a third-party provider. (Basically, it is a crimeware service embedded in the automated exploitation framework.) It also enables the owners of SO to charge more per licence.

We analysed this functionality in SO to understand exactly how the iframe obfuscation patterns are generated. This is important because an understanding of iframe obfuscation will help analysts to dissect the attacks more easily. We simply used the payload '<script>alert(1);</script>' and obfuscated it using the SO iframe cryptor service. Figure 1 shows the output of this service.

The generated obfuscated code adds some '%' characters into a given JavaScript call and declares it as a value to A12836177. Later on, a JavaScript replace call is used to change all the '%' characters to null (''). An additional function is generated, called gd. Then, the code is mixed up with random JavaScript calls to increase its complexity.

Figure 1: The Sweet Orange iframe cryptor in action.

This is a simple example of how SO builds the obfuscated iframes inside the framework.

Domain verification system

SO implements a centralized domain management system. It makes extensive use of domain management APIs for easy operational and functional tasks. The BEP has a built-in domain-scanning engine (Scan4You) which provides information about the state of running and

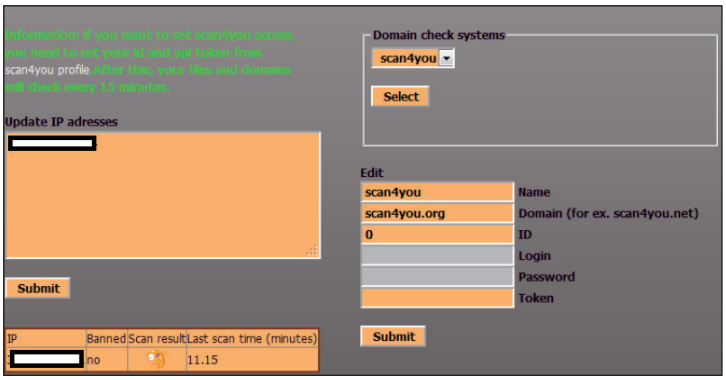

Figure 2: Anonymous service – Scan4You.

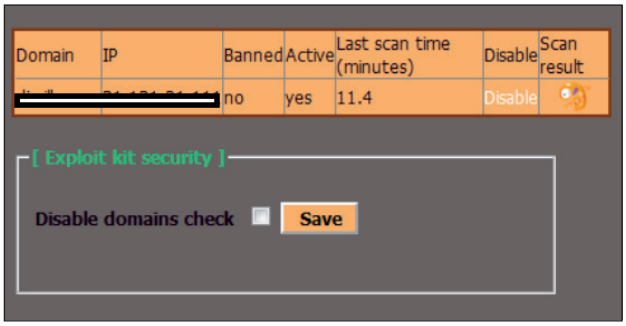

Figure 3: Sweet Orange domain security check.

Supported anti-virus	Supported blacklists
Kaspersky, Solo, McAfee, Bit Defender, Panda, F-Prot, Avast!, Virus Blok Ada, Clam AV, Vexira, Norton, Dr Web, AVG, ESET NOD32, G DATA, Quick Heal, A-Squared, IKARUS, Microsoft Security Essentials Antiviruses, Norman, AntiVirus (Avira), Sophos, NANO, ArcaVir, COMODO, F-Secure, Virus Buster, eTrust, Trend Micro, AhnLab V3 Internet Security, Bull Guard, VIPRE, Zoner AntiVirus, K7 Ultimate.	ZeuS domain blocklist, ZeuS IP blocklist, ZeuS Tracker, Malware Domain List (MDL), Google Safe Browsing (Firefox), Phish Tank (Opera, WOT, Yahoo! Mail), hp Hosts, SPAMHAUS SBL, SPAMHAUS PBL, SPAMHAUS XBL, Malware Url, Smart Screen (IE7/IE8 malware & phishing website), Norton Safe Web, Panda Antivirus 2010, (Firefox Phishing and Malware Protection), SpamCop.net and RFC-Ignorant.Org.

Table 1: Scan4You: list of supported AV and blacklists.

blacklisted domains – it scans the websites that are injected with malicious iframes.

The user can configure the domain-scanning service with username, password and API token. This information is entered in the SO panel (see Figure 2) and once it has been provided a scheduler service is set up that runs scans after a couple of minutes. This process is deployed for active domain verification so that the attacker can perform alter operations if a domain is flagged.

Scan4You [4] is an anonymous service that scans malware against multiple anti-malware products and checks domains against a number of domain blacklists – and crucially, does not report the results back to the anti-malware/blacklist developers. The service is updated periodically to include newer versions of anti-virus software and blacklists. It can thus determine whether the domain hosting SO has been blacklisted or not, and which anti-virus engines can detect the malicious binary. Table 1 shows the list of anti-virus engines and blacklists supported by the service.

As a security measure, the domain scanning function can easily be disabled (see Figure 3). This disrupts the flow of outgoing traffic from the domain hosting the SO panel and allows it to generate a new link (URL) if the previous one has been marked as malicious. No traffic that points to the old link is accepted, and such traffic is discarded by the server running SO.

The domain management API is implemented using the HTTP protocol, which provides easy control over the network simply by sending HTTP requests to fetch the data. Table 2 shows the primary API calls used to gather data from the infected domains.

Based on the information presented in Table 2, an IDS signature can be crafted using the primary command which generates heavy traffic.

Traffic distribution system

Almost all BEPs implement a Traffic Distribution System (TDS) to control incoming Internet traffic based on several characteristics. The SO TDS has the following properties:

- The TDS is capable of filtering traffic and implementing redirection using browser user-agent strings, IP addresses, geo-localization, etc. The traffic can be restricted based on user-agent, installed operating system, type of browser, HTTP content and referrer check by defining filtering rules. In addition, the TDS has built-in load-balancing capabilities.
- It builds statistics based on the incoming traffic and categorizes it into individual IP addresses, number of visits, etc. It also adds password protection and subverts crawlers to gain any information about the hosting server and avoid discovery.
- It has IP timeout functionality that determines the number of times a particular IP can visit the server without being banned. Another functionality is exploit link lifetime management, through which SO minimizes the chances of detection by anti-virus engines.

Function	API and HTTP request
GET current domains	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains
GET AV scan status	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains_av_status
GET AV scan status (JSON)	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=get_domains_av_status&json=1
SET domains	http://[infected IP]/aw/scrt/dmngr.php?key=[value]&a=set_domains&domains=domain1, domain2, domain3

Table 2: Domain management APIs used in Sweet Orange.

Figure 4: Traffic limit in SO.

Figure 4 shows that the maximum traffic limit implemented in SO is 150,000 unique hits.

Advancements in performance

During our analysis, we have noticed a few improvements in SO's request processing mechanism to make the exploitation process faster. This is done to achieve high performance and optimization.

PROPACK

Batch mode execution

The ProPack BEP implements a buffer-based technique to manage incoming connections. The buffer holds information about the victim's machine including what plug-ins are present, the OS version, IP address, etc. When connection attempts are received from target machines, the exploit-serving component initiates a buffer which is used to queue the requests. In other words, ProPack executes batch processing in which all the connection attempts are treated as jobs that are required to be completed without manual intervention. This means that all the specific data is selected earlier and pushed into the exploit-serving component depending on the information extracted from the user's machine. In addition to this, the threading is done efficiently. With proper threading and batch processing, multiple requests can be served at the same time and every thread is shipped with a different executable that is obfuscated differently. This approach also helps to deploy server-side polymorphism, in which

```
alert tcp $HOME_NET 1024: -> $EXTERNAL_NET $HTTP_PORTS (msg:"Propack Exploit Detection"; flow:established,from_
client;
flowbits:set,Propack;
flowbits:noalert;
content:"GET";
http_method;
content:".php?j=1";
 http_uri;
content:"|26|k=";
within:3;
content:" HTTP/1.1|0d 0a|";
within:15;
content:!"|0d 0a|Cookie|3a| ";
http_header;
pcre:"/\.php\?j=1&k=[12345]/U";
reference:url,[ ];  classtype:Exploit; sid:XXXXXXXXX; rev:1; )

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET 1024: (msg:"Propack Malware Binary Successfully Loaded ";
flow:established,from_server;
flowbits:isset,Propack;
content:"Content-Disposition: attachment|3b| filename=";
offset:50;
depth:400;
content:"MZ";
distance:0;
content: "PE|00 00|";
 within:250;
reference:url,[ ]; classtype:Exploit; sid:XXXXXXXXX; rev:1; )
```

Listing 1: ProPack detection signatures.

executable files are generated randomly with different signatures.

Post processing – traffic analysis

ProPack uses the Sypex geo-location library to fingerprint the origin of requests by analysing the IP address of the client. Blackhole uses the MaxMind geo library for processing traffic information based on the IP address. Newer exploit packs are shifting away from using MaxMind to using Sypex because of advantages of the latter such as high speed and low memory consumption. Sypex can easily be integrated with a batch processing routine by implementing caching in memory which increases speed significantly. As Sypex is written in PHP, it can easily be plugged in with the BEP components. Sypex uses binary mode to implement storage structures, avoiding JSON and XML, which consume a lot of processing time. In binary mode, the storage data can easily be differentiated by placing null characters at the end. In order to search for information about IP addresses in the database files, Sypex reads a definite chunk of data from the hard disk, thereby avoiding random searching. For this, Sypex implements a search index using the first byte of the IP address. The idea is to traverse less data to find the requisite information and increase the speed. Following our analysis of ProPack traffic, Listing 1 shows possible network signatures that can be used to detect the ProPack exploit kit.

CONCLUSION

In this paper, we have explored some of the basic design advancements in the Sweet Orange and ProPack exploit packs. Understanding the design of these exploit kits allows analysts to dig deeper into the new methods used by these exploit kits to infect systems. We can expect further developments in these exploit packs in the near future.

REFERENCES

[1] Sood, A. K.; Enbody, R. J. Browser Exploit Packs – Death by Bundled Exploits. http://secniche.org/presentations/virus_bulletin_barcelona_2011_adityaks.pdf.

[2] Just the Sweet Orange. http://malware.dontneedcoffee.com/2012/12/juice-sweet-orange-2012-12.html.

[3] The Exploit Distribution Mechanism in Browser Exploit Packs. http://magazine.hitb.org/issues/HITB-Ezine-Issue-008.pdf.

[4] Scan4You Anonymous Service. http://scan4you.net/.

TUTORIAL

SHELLCODING ARM: PART 2

Aleksander P. Czarnowski
AVET Information and Network Security, Poland

In the first part of this series [1] we discussed the basic background information needed to understand the principles of ARM shellcoding. In this follow-up article we will dissect some previously crafted shellcode.

THE GETPC PROBLEM

The shellcode techniques we've discussed so far have a couple of requirements:

- The code must be position independent (PIC).
- The shellcode data (such as parameters for syscalls) must be positioned at the end of the code section.

This raises the issue of how to determine the Program Counter (PC) value. This value can be used to calculate the offset to the shellcode data and other crucial areas such as encrypted code (this will be discussed in more detail in the next article).

Figure 1 shows the most basic shellcode layouts:

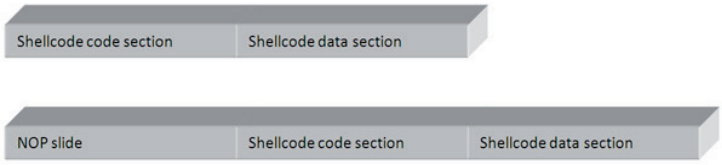

Figure 1: Basic shellcode layouts.

What is missing from Figure 1 is a return address, but since this section is random in the sense that it changes from vulnerability to vulnerability (and even between system revisions), we can't predict it, and it is outside the scope of this article.

To better illustrate the GetPC problem, let's compare x86 shellcode techniques with ARM ones.

In x86 architecture, the two most popular 'GetPC' constructions are:

- JMP/CALL/POP reg trampoline code
- Use of FSTENV

As shown in Table 1, the trampoline code is quite simple. The POP ECX instruction returns the EIP value, which is a pointer to the shellcode data section since the address pushed onto the stack by the CALL instruction points to the next instruction after the CALL opcode. However, in our case there is no valid code there, just data.

Address	Instructions
0	JMP start
+5 (rstart)	POP ECX
[…]	Rest of the shellcode
Start	CALL rstart (+5)
start+5	Shellcode data section

Table 1: Trampoline code.

One might wonder why, besides the pointer to the shellcode data section, we need the first JMP instruction. The reason is bad bytes. Consider the following code:

```
CALL  $+4
POP   ECX
```

The call instruction will be assembled as:

```
E800000000
```

There are clearly too many bad bytes to deal with such opcode in the case of shellcode.

The second trick is based on the FPU instruction FSTENV, which saves the FPU and part of the CPU state in memory. In protected mode, 28 bytes of memory are needed to store the saved state:

Address	Instructions
0	FLDZ
+2	FSTENV SS:[ESP-0xC]
+6	POP ECX

Table 2: The FPU instruction FSTENV saves the FPU and part of the CPU state in memory.

After the code shown above has been executed, the ECX register contains the address of the FLDZ instruction.

It is worth mentioning that both methods are system-independent, unlike methods based on Structured Exception Handling (SEH) which only work under *Windows*, for example. It should not come as a surprise, therefore, that ARM shellcode can also be written in such a way that enables execution under different operating systems. Obviously the API calling convention changes from platform to platform, but the shellcode framework can be reused in such cases.

So how is it done on the ARM platform? There are a number of features of ARM architecture that particularly appeal to shellcode authors – one of which is the ability to switch between ARM and Thumb modes and the fact that this process does not require any special preparation (unlike switching between real and protected mode on x86 CPU, for example). Why is this feature so important to shellcode authors? Since the Thumb/Thumb2 instruction set is 16 bits long, the instruction encodings are not only shorter (shorter shellcode means more flexible and more reliable shellcode), but as a side effect, many bad bytes are eliminated. We will discuss this in more detail later in the article.

API CALLING CONVENTIONS

To understand all the shellcode presented here we first need to understand the *Linux* API calling convention, which is a reflection of the ARM calling convention.

Let's start with the *Linux* execve() calling structure:

- R0 must point to the '//bin/sh' string
- R1 must point to the '//bin/sh' string

Address	Bytes	Instructions	Comment
0	e28f6001	add r6, pc, #1	This is an ARM-type GetPC construction based on jump. The BX instruction not only sets PC to the R6 value, but also switches ARM into Thumb mode.
+4	e12fff16	bx r6	
+8	4678	mov r0, pc	This is the second part of the GetPC construction – now R0 contains the current offset of the shellcode. Note that from this point on, the shellcode is executing in Thumb mode.
+A	300a	adds r0, #10	The R0 register value is adjusted to point to the data section (R0 points to the +16 address) – points to //bin/sh string.
+C	9001	str r0, [sp, #4]	The section data pointer is placed on the stack.
+E	a901	add r1, sp, #4	R1 = SP+4 – points to the //bin/sh string.
+10	1a92	subs r2, r2, r2	The R2 register is zeroed out (R2 = 0). Subs r2, r2, r2 is used in order to avoid bad bytes.
+12	270b	movs r7, #11	R7 contains the *Linux* SYSCALL number (0x0B = execve).
+14	df01	svc 1	*Linux* SYSCALL.
+16		//bin/sh	Data section for execve SYSCALL.

Table 3: Shellcode instructions.