Towards Secure Scripting Development

Facundo Ferrer
facundo.ferrer@intel.com

Andres More
andres.more@intel.com

Argentina Software Development Center (Intel)

Abstract

This work reflects two separate efforts towards implementing security awareness in projects being developed using scripting languages. The same development process was used in both projects which included a specific focus on security. Besides summarizing lessons learned, this work recommends automatic tools for static code analysis to ensure that usual pitfalls are avoided from the very beginning. This work contributes with an overview of specific insights on Perl and Python security, together with a list of publicily available support tools, and how they are best applied and integrated into the development process. The proposed development framework is based on a supporting infrastructure which minimizes effort on peer reviews and allows early identification of security issues, making their resolution easier and cheaper.

1 Introduction

This works shows the experience on how scripting development can be done with a security focus. Besides best known methods on secure development, specifics about the Perl and Python programming languages are discussed in detail.

1.1 Background

Among the different programming languages being used around the world, scripting languages account for a significant portion. As of today, Python holds the 6th and Perl the 9th position on the ranking of the most popular languages according to TIOBE. See figure 1 for the April 2011 TIOBE index.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 42

April '11	April '10	Delta	Language	Rating
1	2	↑	Java	19.043%
2	1	↓	C	16.162%
3	3	=	C++	9.225%
4	6	↑↑	C#	7.185%
5	4	↓	PHP	6.584%
6	7	↑	**Python**	4.931%
7	5	↓↓	Visual Basic	4.682%
8	11	↑↑↑	Objective-C	4.386%
9	8	↓	**Perl**	1.991%
10	10	=	Javascript	1.512%

Figure 1: TIOBE Index of Programming Languages (April 2011)

Scripting languages are popular because their use does not requires of a compiler or any complex toolchain, as they are interpreted on-the-fly at runtime. In the case of major GNU/Linux distributions, the basic script runtimes are available by default with their base installation.

Scripting languages are multiple purpose. Besides to have a low learning curve and to be easily found, they glue different system services and applications, taking advantage of the UNIX philosophy of providing independent tools and filters. Scripts are wonderful tools to do task sketching, allowing early identification of next steps by using a throw-away prototype.

Moreover, as scripts can be seen as a set of commands in a text file, it is easy to learn from the examples available on the system. Almost by definition, scripts files are readable by the user executing them. People find useful the idea of automating their iterative tasks in a way similar to what is done manually on the command line console.

1.2 Problem Definition

Scripting programming languages such as Perl and Python are mostly used at system level to coordinate services and even to implement administrative tools such as packaging systems. They are also the preferred choice to implement tasks to be done during system startup and shutdown. These tasks require the enforcement of privileges as their action will impact on the operating system behavior; affecting any application, service or data.

When running with privileges at system level, any wrong input or unhandled scenario may drastically impact the system. For instance, any malicious input may be executed as a system command with administrative rights.

1.3 Related Work

Secure coding in scripting languages has been considered worldwide by different people [2] [3] [4]. The development of supporting tools [5] and even the inclusion of new security features on the scripting languages is also work in progress [6] [7].

2 Secured Software Development Process

2.1 Security Oriented Activities

Before entering into the specific details for scripting languages, there are several considerations which apply to the stages of the development process itself. These considerations follow the Security Development Lifecycle, which was developed at Microsoft to save costs and resources during development [8]. This process specifies security concepts including the usage of checklists, tools to double check security threats, security oriented peer reviews and specialized testing.

Development Stage	**Security Activity**
Definition	Training
Architecture	Design Patterns
Design	Design Reviews
Early Implementation	Automated Support
Final Implementation	Peer Reviews
Testing	Specialized Testing

Figure 2: Security Oriented Activities

An overview of the security related activities per process stage is shown at figure 2. Although similar to the usual development activities for quality assurance, the list considers specific security oriented topics. During project definition, the development team as a whole needs to review security topics including Writing Security Requirements, Secure Architecture, Secure Design, Secure Coding and Testing Security. Investing some hours on informal discussions around the security features and pitfalls of the technology to be used for development will save a lot of headaches later on.

Once hands-on development has started, Architecture and Design should consider the usage of secure design patterns [9], having explicit reviews with focus on security aspects such as authentication and authorization. While starting the coding phase, the team needs to put enfasis on the establishment of automated support to track product changes, and before concluding implementation a detailed review of critical sections of the codebase needs to be completed.

In the case of iterative development, we can clasify the security activities according to their occurence as shown in figure 3.

Basis	**Security Activity**
Once	Training, Automated Support
Every Iteration	Gap Analysis, Peer Review, Testing
As Required	Architecture and Design Reviews
Major Release	Penetration Testing

Figure 3: Security Oriented Activities with Iterative Development

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 44

2.2 General Considerations

2.2.1 Best Known Methods

During every phase of a development process, security checklist needs to be available and completed. Once in *firefighting mode*, it is inevitable that the urgency will trump importance. Keeping a hardcopy of the activities and their expected deliverables is a valuable asset for the team.

In the case of security, it is really important to keep focus and a critical mindset during the end-to-end product development. Checkpoints should be established to review development milestones with recognized security experts when available. The key milestones to consider are usually the completion of each stage: definition, architecture, design, coding and testing.

Design reviews with focus in security are other important discipline to follow. Each time complex features or bug fixes are faced, consideration must be taken to check if they might deserve security reviews. In particular this should be done every time the design is modified; with special emphasis if the design is being done for the first time.

Code peer-reviews with focus in security should also be done by all the development team, including the testing and quality assurance members. Critical sections handling authorization or privileges should be carefully reviewed every time they are updated.

If the product execution handles significant parsing of data, enforcing automated fuzzy testing of malformed data should be mandatory; bugs on input validation came from the abuse of regular expression features. Fuzzy testing allows the team to exercise error handling and input validation by using randomly generated information.

All sources of the project deliverables should be kept over version control in order to avoid loosing effort due to human mistakes and to be able to reproduce a reported issue with an old version of the code.

2.2.2 Static Analysis

From the efficiency point of view, the development team should remove low priority issues in the code as soon as possible. Wrong indentation, spellchecking and other cosmetic findings should be automatically detected and fixed early during development, not during peer review or testing. In this way, the effort required to solve them is substantially lower and hence frees up effort for other tasks and activities adding more value to the product being developed.

To enforce automatic analysis of code, static analyzers should be used. An ideal static analyzer tool shall have the following characteristics:

1. **configurable**: the execution of rules shall be configurable; allowing the user to decide which ones will be enforced.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 45

2. **severity levels**: the rules should be split into different levels and the levels should be selectable at runtime; allowing to easily categorize and prioritize the findings when applying the tool for first time.

3. **extensibility**: the rules should be easily extended; allowing the team to include new ones when analyzing the outcome of peer reviews to identify usual issues to avoid in the future.

4. **deterministic heuristics**: if the tool applies heuristics to determine an improvement opportunity, it must not show different results over the same set of files.

3 Perl

3.1 Security Overview

Perl was developed by Larry Wall [10] as a general-purpose scripting language mostly targeting the processing of text. It is used mostly at system level. The overall syntax derives from the C language which allowed significant popularity.

Regarding security at Perl, the basic starting point is the perlsec manual page. The discussion there defines vulnerability contact information and analyzes main mechanisms and concerns to keep in mind during development.

Perl provides a built-in mode to reduce security threats due malicious input. This taint mode enables an enforcement of input validation and issues a warning at runtime unless the data is sanitized with a regular expression or one of several built-in functions. All input which comes from the user is required to be untainted explicitly.

The enforcement approach is conservative, if any expression contains tainted data, then any sub-expression is hence considered tainted as well. Noteworthy, any operation on strings that uses localization information is considered tainted, as it is possible for a localization definition to be untrustworthy.

As the scripts will use the inherited environment, it is usually recommended to include proper cleaning and checking of the `PATH` and `LD_LIBRARY_PATH` environments as part of the initial setup. Otherwise, a malicious definition might led to unwanted results when adquiring privileges during execution.

When using setuid to automate system administration tasks requiring privileges, it is adviced to drop privileges when they are not longer required, and even to reject such grant when the task will not use those rights.

3.2 Supporting Tools

As in many other languages, coding style is a small but important code property. It allows the peer reviews to go smoothly, only focusing on functional things. The perltidy [1] tool provides a non error-prone approach to solve the issue.

[1] Is a Perl script indenter and reformatter, available at `http://perltidy.sourceforge.net`.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 46

Moreover, the perlcritic [2] tool attempts to identify awkward, hard to read, error-prone, or unconventional constructs in the code. Most of the rules enforced follow the *Perl Best Practices* book [11]. Thus, clearly documented diagnostics, rationale, examples and resolution steps are provided on each finding.

The perlcritic tool enforces more than 150 complex rules, each of those belonging to different themes such as: bugs, complexity, core, cosmetic, maintenance, pbp, performance, portability, readability, security, tests and unicode.

Both tools are extensible, provides documentation and diagnostic information on findings, and are designed to be extensible. They have Open Source licensing so they functionality can be reviewed under the hood.

A simple example of perlcritic's output is the following:

```
> perlcritic --brutal --verbose 10 FILE
Return value of flagged function ignored - print at line 681, 5.
  InputOutput::RequireCheckedSyscalls (Severity: 1)
    This performs as InputOutput::RequireCheckedOpen/Close
    except that this is configurable to apply to any function,
    whether core or user-defined.
    If your module uses Fatal, Fatal::Exception, or the autodie
    manpage then any functions wrapped by those modules will not
    trigger this policy. For example:
        use Fatal qw(open);
        open my $fh, $filename;  # no violation
        close $fh;               # yes violation
        use autodie;
        open $filehandle, $mode, $filename;   # no violation
    Currently, autodie is not properly treated as a pragma; its
    lexical effects aren't taken into account.
```

4 Python

4.1 Security Overview

Python was created by Guido van Rossum [12] as a high-level general-purpose programming language with the inclusion of high-level data types. This programming language has a wide range of uses: it is used for teaching computer science, as a domain-specific language, and for developing large applications. Part of its popularity stems from the fact that Python is an interpreted language with its own virtual machine [17].

Something that all Python users have in common is the belief that the code they are executing in the Python interpreter is not malicious. However, the current version of Python lacks any mechanism to enforce a security policy related to controlling access to resources. While some informal methods exist,

[2] Is a command-line interface to critique Perl source and identify opportunities of improvements, available at `http://www.perlcritic.org`.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 47

Python lacks any mechanism built into the language or its implementation to prevent code from accessing resources such as files or sockets [17].

The Python developer team implemented a *restricted mode* similar to Perl's tainted mode through the *rexec* and *Bastion* modules but starting in version 2.3 of Python, both modules were disabled[17]. Some years after, Python Security started as an individual Open Web Application Security Project (OWASP) initiative with the mission of making Python the most secure programming language in the world [14]. This is a good starting point of Python security related topics.

Python does not come with built-in support for input validation but there are some efforts to implement taint mode in Python [6] [7]. The approach described by Kozlov et al. [6] implies the modification of the Python interpreter, while the Conti et al's s approach [7] describe a library based taint mode.

According to OWASP research, the most efficient way of finding security vulnerabilities is manual code review. As this activity is very time-consuming, requires high expert skill, and is prone to overlooked errors, the security community is actively developing automated approaches to finding security vulnerabilities [6].

There are two approaches to automatic security analysis, dynamic and static analysis. The dynamic analysis is the process of extracting program's properties by executing and monitoring it under different input data.

On the other hand, static analysis is the process of extracting program's properties from the source code solely, without executing it. For the first approach, the implementation of a Taint mode as described earlier is a good way to reduce the risks associated with input data. For static analysis, there are some tools that could give a good help during the development process.

4.2 Supporting Tools

`pylint` is a tool developed at Logilab.org [18] to perform static code analysis. It has a very complete set of tests and allow custom test development. The tests are shown under different categories, including refactor, convention, warning, or error.

The following example shows pylint in action, identifying comments, warnings and errors.

```
> pylint FILE
C:081: Line too long (85/80)
W:567: No exception type(s) specified
E: 86: myexit: Undefined variable 'mylib'

Your code has been rated at 4.25/10
```

5 Supporting Infrastructure

An automated and unattended check used as an early feedback system allows easy and cheap resolution of issues. The supporting infrastructure should apply automated acceptance tools including security checks every time the main development trunk is modified. Ideally they should include the same regression tests or unit tests being used during coding and testing will also help to reduce the issues detected during the review stages.

In the case of legacy code, a convergent approach migth be the best fit. Before entering peer reviews or testing phases, the number of findings need to be reduced and no critical or high exposure issues should be reported.

A pseudo-code version of the continuous integration script to be used during development or as testing acceptance criteria is the following:

```
# process files if they match the expected suffix
FILES=src/*.{pl,pm,py}

# check if indentation is OK
foreach file in FILES; do
        indent file > tmp && diff orig new && echo COSMETIC-FAIL
done

# run unit test cases
make unit-test && echo UNITTEST-FAIL

# run regressions tests
make regression && echo REGRESSION-FAIL

# gather previous quality level
svn -r PREV FILES > /dev/null;
OLD=‘static-check FILES | wc -l‘;
svn up > /dev/null
NEW=‘static-check FILES | wc -l‘;

# only pass if quality has improved
if (NEW > OLD) then echo QUALITY-FAIL;
echo OK;
```

This script can be used by both developers and testers besides to be enforced automatically after each code change. The historical execution figures on execution time and finding count can be reviewed periodically to easily understand current gaps and guide next steps in bug fixing.

In our experience this approach allows project development to be streamlined after applying this type of solution, as peer reviews are done faster, reporting minor comments. Also, quality findings are decreased continuously if automated measuring tools are incorporated. Figure 4 shows the historical data in one internal project.

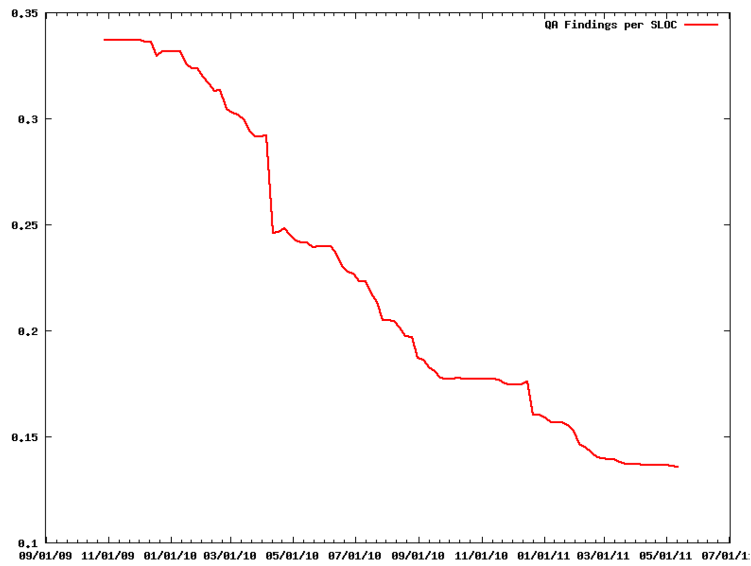

Figure 4: Total Static Analysis Findings

The whole set of regression tests being executed together with the static analyzer provides an important indicator of the project status over time, see Figure 5. It can be also seen how the execution time is increased due the constant addition of checks after each development iteration, time when the team identify common pitfalls after code reviews are completed.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 50

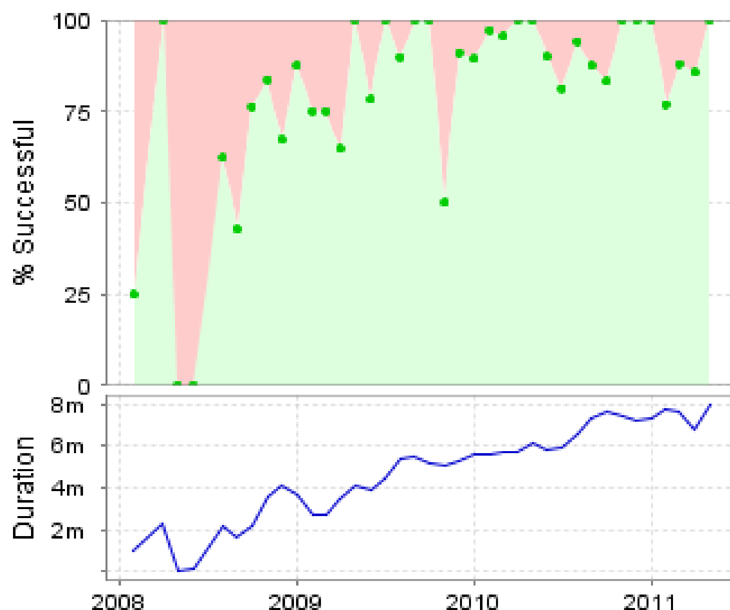

Figure 5: Successful Ratio and Duration of Automated Tests

6 Conclusions

This work shows how to avoid common security pitfalls when developing system-level scripts. The discoveris are done early in the development process and without pain thanks to the application of predefined processes and automated tools.

By following a documented development process with focus in security, the likelihood of issues ocurrence will be highly reduced regardless deadlines and required overwork due changed requirements or deadlines.

An ideal approach is to make sure that specific training in security is scheduled and taken by the whole product development team, not only on the development but also the testing side. Another ideal requirement is to have automated support to identify coding issues early.

Automated support must be enforced, since one automated test is far more useful than ten manual tests as it can be used during the whole development cycle, catching issues earlier and making them cheaper to be fixed. This technique automated tests can easily include no-brainers on spellchecking, style and best-known-methods that otherwise are too painful to resolve during a quality gate at a later stage.

Static analysis is also a practical and efficient approach, as it will let the team to have less discussions on low priority findings (such as cosmetics ones) and save effort to focus on relevant issues.

40JAIIO - WSegI 2011 - ISBN: 978-987-1312-22-1 - Página 51

This work also shows how to automate product builds by enforcing the checks on every product change. With this approach, the product only suffers controlled changes and the product health is constantly monitored.

7 Further Work

A security analysis with focus on productivity may be done on other scripting languages. Shell scripting is an interesting candidate to start with, due its pervasive presence at system level.

Besides the tools previously discussed there are more static analysis tools available such as RATS from HP-Fortify [19]. Also, most of these tools are easily expanded through plug-ins and including new types of check rules is a potential next step once the product development team is familiar with the tool.

8 Acknowledgments

The authors would like to recognize the Argentina Distributed Computing Solutions team working at Intel Software Argentina for their contributions and reviews. Many of the ideas also came from the Argentina Security Development Lifecycle Forum gathering security representatives (and enthusiasts) from the different projects on the site.

References

[1] TIOBE Software. TIOBE Programming Community Index. Retrieved April 2011. `http://www.tiobe.com/index.php/tiobe_index`.

[2] Okubo. Takao, Tanaka. Hidehiko, Secure Software Development through Coding Conventions and Frameworks, IEEE, 2007.

[3] K. Mazurak, S. Zdancewic, ABASH: finding bugs in bash scripts, Univeristy of Pennsylvania, 2007.

[4] P Fenwick, J. Richardson, Perl Security, Perl Training Australia, 2008.

[5] Static Detection of Security Vulnerabilities in Scripting Languages. USENIX

[6] Kozlov D., Petukhov A., Implementation of the Tainted Mode approach to find security vulnerabilities in Python. Moscow State University, 2006.

[7] Juan Jose Conti, Alejandro Russo, A Taint Mode for Python via a Library. Universidad Tecnologica Nacional, Facultad Regional Santa Fe, Argentina. Chalmers University of Technology, Sweden.

[8] Michael Howard and Steve Lipner, The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software, Microsoft Press, ISBN: 0735622140, June 2006.

[9] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama, A survey on security patterns, Ritsumeikan University, 2005.

[10] Wall, Larry, Tom Christiansen and Jon Orwant (July 2000). Programming Perl, Third Edition. O'Reilly Media. ISBN 0-596-00027-8.

[11] Damian Conway. Perl Best Practices. O'Reilly Media (July 19, 2005). ISBN: 978-0596001735

[12] Lutz, Mark (2009). Learning Python (4th ed.). O'Reilly Media. ISBN 978-0596158064.

[13] Practical PERL for Security Practitioners. SANS. 2004.

[14] Python Security website, part of the OWASP project, retrieved April 2011, `http://www.pythonsecurity.org/`.

[15] OWASP Python Static Analysis Project, `https://www.owasp.org`.

[16] R. Austin, R. Schlesinger, B. Setzer, A Security Scripting Course, Kennesaw State University, ACM, 2005.

[17] Brett Cannon, Eric Wohlstadter, Controlling Access to Resources Within The Python Interpreter, University of British Columbia.

[18] Logilab.org. Logilab - informatique scientifique et gestion de connaissances. Retrieved April 2011. `http://www.logilab.org/`

[19] Rough Auditing Tool for Security (RATS). Fortify Software, an HP Company.