The evolution of a security control

Olgierd Pieczul[1,2] and Simon N. Foley[2]

[1] Ireland Lab, IBM, Dublin, Ireland.
[2] Department of Computer Science, University College Cork, Ireland.

Abstract. The evolution of security defenses in a contemporary open-source software package is considered over a twelve year period. A qualitative analysis style study is conducted that systematically analyzes security advisories, codebase revisions and related discussions. A number of phenomena emerge from this analysis that provide insights into the process of managing code-level security defenses.

1 Introduction

During an application's lifecycle, the ideal is that there is a continuing process for vulnerability discovery and repair. A more pragmatic viewpoint of the process is that vulnerabilities are unknowingly introduced (or re-introduced) during code maintenance, existing vulnerabilities are missed, misreported and misinterpreted and repairs to defenses are incomplete. Even when a threat is clearly identified it can be a struggle to provide an adequate security defense. PHP's "safe mode" as a means to provide security for multihosting is a case in point; ever since its introduction, new ways of bypassing its latest defense/repair have been repeatedly discovered, eventually leading to the decision to remove it from PHP. One might argue that the difficulty in providing an adequate mechanism in this case is attributed to a flawed design and the relatively large scale of the software. However, this kind of problem can also be observed at a smaller scale and in simpler code that implements routine tasks. The Shellshock vulnerability in the Bash shell allowed custom code execution due to a bug in code parsing the environment variables. The fix was released quickly in 2014, however, it took five further vulnerabilities and remediations before the issue was believed fixed.

Research aimed at gaining insights into these kinds of issues has tended towards quantitative studies. Metrics such as number of bugs over time, rate of appearance, type and severity, can be gathered and statistically analyzed [1, 2]. Such measurements can point to interesting trends, however, they rely on their hypothesis and the efficacy of the underlying data which, for example, may include attributes such as CVSS scores and lines of code. Furthermore, it cannot help one understand why a particular security weaknesses persists over time and cannot be properly addressed, nor what causes the implementation of a weak security mechanism. While a quantitative study can provide a basis for supporting a hypothesis, where a hypothesis does not form the basis of the research question an exploratory approach is informative.

In this paper we take an exploratory approach to gaining insights into these issues. Informed by qualitative research techniques, we carried out a systematic study of the evolution of a security control over a long period of time with a view to discovering security-relevant phenomena that emerge. The paper is organized as follows. Section 2 outlines the methodology that was followed during the study. The study is based on a security control in Apache Struts and Section 3 provides the background necessary to understand the technical account given on the evolution of this control in Section 4. Section 5 discusses a number of phenomena that emerge during the analysis of this control.

Note for the reader/reviewer. In as much as possible, the paper has been organized so that much of the detailed technical accounts of the vulnerabilities and attacks in Sections 3 and 4 may be skipped by a reader who has a greater interest in the emergent phenomena than in the low-level details of Struts.

2 Methodology

We based our study on Apache Struts, a popular web application framework for Java, developed under the auspices of the Apache Software Foundation. Our rationale for selecting Struts as a good representative of contemporary software, is as follows. Struts is a mature and widely used package that has been developed according to best practices, both in terms of code implementation as well as development life cycle, with a documented policy and change management process. The security processes surrounding Struts are transparent and include documented processes for reporting vulnerabilities and publishing security advisories. We focussed our attention on the functionality of one particular Struts security control that has had a series of reported security vulnerabilities and has evolved over time. The chosen control is sufficiently critical to ensure both internal and public interest in identifying security problems, and that reported issues are treated seriously by the development team.

We performed a systematic analysis of the Struts source code published over twelve years from 2004 to 2015. This was done in a qualitative style, whereby the objective was to identify security-related phenomena, or patterns, that emerge from the activity of making code revisions. The analysis focused on the code revisions that arose as a consequence of, and/or were the cause of, the security advisories over that period. In particular, these were related to a security-control that is responsible for preventing the injection of malicious code into the framework via the parameters of web page requests. This security-control checks parameter values passed to the Struts `ParametersInterceptor` and `CookieInterceptor`, preventing their misuse. Note, that the `ParametersInterceptor` functionality originated in the XWork project and was later merged into Struts; the XWork source was subject of the analysis during this intial period.

We reviewed: the security advisories/vulnerability publications; code-updates (security-related or otherwise); related discussions on the development mailing list, and other publications often contributed by the vulnerability reporters who

sometimes provided additional technical details. Often, only partial details of an attack were published, and in all cases it was possible to re-construct/implement the attacking code by reverse-engineering the code changes and published information. This led to the analysis of an estimated 300+ security-relevant code changes over the evolution of Struts.

In carrying out this analysis we identified the code changes that had an impact on security, either fixing a known vulnerability or introducing a security issue. For ease of exposition, the analysis is summarized in terms of aggregate changes over releases that culminate in a published security-related release, with one exception. In this way we believe that our inferences about the developer's intentions are more reliable than those based on (possibly incomplete) changes made in between security releases. While the observation of changes in-between the releases may provide an insight into security mechanism evolution, it was not clear whether the changes at these stages could be considered complete. As a result we discovered 20 key security related changes, presented by row in Table 1.

During our investigation we identified elements of the security mechanism and mapped the changes into the corresponding categories, give by the right-hand columns in the table. The identified changes often take a simplified form, of regular expression or an acronym, representing the essence of the change. For simplicity, the significant element of the change is highlighted using **bold** text. Note that the categories discovered during analysis are not related to the actual structure of the code, as the corresponding security mechanisms were routinely moved/refactored within the source code, given different names and so forth.

Throughout this process we strove to make observations about vulnerabilities, repairs and coding activities, based solely on the evidence in this corpus.

3 Struts operation

This section provides an overview of those parts of Struts that are required to understand our analysis of the security control used in the study. Struts is a mature and popular web Model-View-Controller framework for Java. One of the features of Struts is the ability to easily separate the business logic from the operational details related to processing HTTP requests. For example, consider a sample piece of code of a web application responsible for handling a request to add an application user by an administrator, presented at Figure 1. The listing shows three parts of the application: class `User` encapsulates the details of an application user; class `AddUser`, implemented as Struts *action*, provides the logic adding user to the system, and the JSP provides the fragment of the view *view* (a page is presented when the action is complete).

Note that `AddUser` does not contain any web-specific logic, which is handled by Struts. It defines getter and setter methods (`getNewUser` and `setNewUser`, respectively) for retrieving and setting the user object, and a `setSession` method (required by `SessionAware`) interface for Struts to set the Session object. For example, client may send a request such as: `http://application/user/add?newUser.name=john&newUser.role=support` in order to add new support user. The request

```
public class User {
    private String name;
    private String role;
    ... // getters and setters for fields

    public boolean isAdmin() {
        return (role.equals("admin"));
    }
}

public class AddUser extends ActionSupport implements SessionAware {
    private User newUser;

    public User getNewUser() {
        return newUser;
    }

    public void setNewUser(User user) {
        this.newUser = user;
    }

    public void setSession(Map session) { // for SessionAware
        this.session = session;
    }

    public String execute() throws Exception {
        if (session.get("user").isAdmin()) {
            DAO.add(newUser);
            return SUCCESS;
        }
    }
}

<s:property value="#session['user'].name"/> added user <s:property
value="newUser.name"/> with role <s:property value="newUser.role"/>
```

Fig. 1: Sample MVC struts application

is received by Struts which, based on its configuration, decides whether it should be processed by a `AddUser` action. It instantiates the class and then (as it implements `SessionAware` interface) calls `setSession` method with a session map for current user. Struts then parses the parameters, creates a new User object using the values from the request, and provides it to the action using the `setNewUser` method. Subsequently the `execute` method executes taking advantage of all the properties that have been set. After the user is added, the JSP page is rendered which in turn refers to action properties such as newly added user and session.

3.1 OGNL

Struts uses Object-Graph Navigation Language (OGNL) [3], an expression language used to get and set properties of Java objects. OGNL expressions are evaluated against a collection of objects called *context*. One of the objects, called *root*, is distinguished as the default root of the object graph. When processing a request, Struts sets the current action object (for example `AddUser`) as the root. In the example in Section 3 the expression `newUser.name` is used both in request parameters and also in the JSP file. The values are being accessed through public getters and setters. For example, the OGNL `newUser.name` to get the name of an object is equivalent to Java code `action.getNewUser().getName()`. Similarly, using OGNL to set a value expressed as `newUser.name` to `alice` is equivalent to Java code `action.getNewUser().setName("alice")`.

Other Struts-specific objects such as session, request and application configuration are also included in the OGNL context. For example `#session['user'].name`, expression can be used to access a `name` property of an object that exists in session map under index `'user'`. Finally, context contains number of variables that control OGNL behavior, such as rules for accessing classes depending on type, access restrictions, caching and so forth. Figure 2 provides an overview of context structure at the time of execution of `AddUser` action.

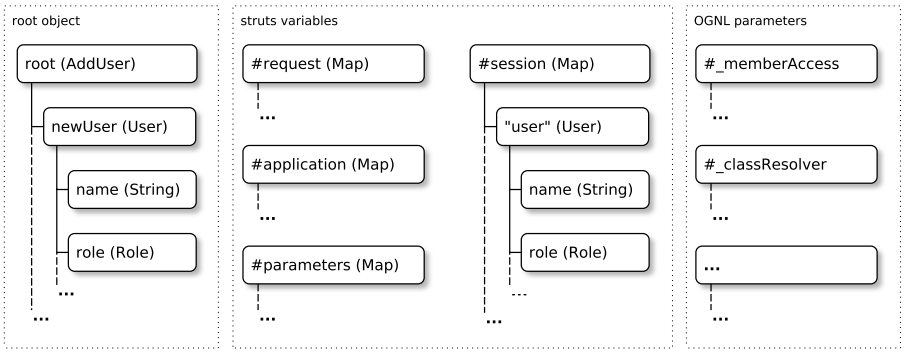

Fig. 2: OGNL context in example application

3.2 Struts Interceptors

Struts *interceptors*, upon which our study is based, parse request parameters and set corresponding values in action objects. Interceptors in Struts are responsible for handling common tasks before/after the action is executed. Typical tasks performed by interceptors is handling HTTP requests (such as request parameters or cookies), input validation, access control, caching and so forth.

Processing request parameters is performed by `ParametersInterceptor`. Our study started on the codebase published at the beginning of 2004, with the release of the XWork 1.0 library in which the interceptor was first implemented and the later merged into Struts. Since then, the functional requirements of the parameter interceptorhave not changed. It iterates over each parameter and sets the action values using parameter name as OGNL expression to identify the object and parameter value as the value to set. Starting from June 2007 `CookieInterceptor` sets action properties based on HTTP cookies.

4 Tracing the evolution of a security control

The parameters and cookie interceptors allow clients to provide a custom OGNL expressions that are evaluated by Struts. OGNL expressions can result in the execution of custom code which accesses program variables. From its first release, this functionality has been considered a security threat and, a mitigating

security control has always formed a part of its implementation. In this section we systematically trace the evolution of this control over a 12 year period: 2004–2015. The results are summarized in the Table 1 and described in detail in the reminder of the section.

date	accepted parameters	accepted cookies	excluded patterns	OGNL
Jan 2004	[empty]	n/a	n/a	**ME**
Dec 2004	excluded: {'=', ',' ,'#'}	n/a	n/a	ME
Feb 2007	excluded: {'=', ',' ,'#', ':'}	n/a	**dojo\..***	ME
Jun 2007	excluded: {'=', ',' ,'#', ':'}	[empty]	dojo\..*	ME
Jul 2008	excluded: {'=', ',' ,'#', ':'}	[empty]	dojo\..*	ME, **SM**
Jul 2008	ex: {'=', ',' ,'#', ':', **\u0023**}	[empty]	dojo\..*	ME, SM
Oct 2008	**[\p{Graph}&&[^,#:=]]***	[empty]	dojo\..*	ME, SM, **SC**
Aug 2010	**[a-zA-Z0-9\.\]\[\(\)_'\s]+**	[empty]	dojo\..*,^struts\..*	ME, SM, SC
Dec 2011	[a-zA-Z0-9\.\]\[\(\)_'\s]+	**[a-zA-Z0-9\.\]\[_'\s]+**	dojo\..*,^struts\..*	ME, SM, SC
Dec 2011	**[a-zA-Z0-9\.\]\[\(\)_']+**	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*	ME, SM, SC
Jan 2012	**\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))***	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*	ME, SM, SC, **EE**
Apr 2012	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*,**^session\..*,^request\..*,^application\..*,^servlet(Request|Response)\..*,parameters\...***	ME, SM, SC, EE
Aug 2012	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))* **[100]**	[a-zA-Z0-9\.\]\[_'\s]+	dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*,^servlet(Request|Response)\..*,parameters\...*	ME, SM, SC, EE
Mar 2014	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**^class\..***,^dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*,^servlet(Rrequest|Response)\..*,^parameters\..*,^action:.*,^method:.*	ME, SM, SC, EE
Apr 2014	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**(.*\.|^\.*|\[('|"))(c|C)lass(\.|('|"))]\[\]).***,^dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*,^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
Apr 2014	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	**default:** (.*\.|^\.*|\[('|"))class(\.|('|"))]|\[\).*; **params:** ^dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*,^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
May 2014	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['\w+'\]\)|(\('\w+'\)))* [100]	[a-zA-Z0-9\.\]\[_'\s]+	(.*\.|^\.*|\[('|"))class(\.|('|"))]|\[\).*,^dojo\..*,^struts\..*,^session\..*,^request\..*,^application\..*,^servlet(Request|Response)\..*,^parameters\..*	ME, SM, SC, EE
Dec 2014	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['(\w|[\u4e00-\u9fa5])+'\])| (\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	**config/params:** ^action:.*,^method:.* (priority) **default:** (.*\.|^\.*|\[('|"))\bclass(\.|('|"))]|\[\).*,(^|.*#),dojo(\.|\[\).*,(^|.*#)struts(\.|\[\).*,(^|.*#)session(\.|\[\).*,(^|.*#)request(\.|\[\).*,(^|.*#)application(\.|\[\).*,(^|.*#)servlet(Request|Response)(\.|\[\).*,(^|.*#)parameters(\.|\[\).*,(^|.*#)context(\.|\[\).*,(^|.*#)_memberAccess(\.|\[\).*	ME, SM, SC, EE, **EC**
May 2015	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['(\w|[\u4e00-\u9fa5])+'\])| (\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	(^|.*#)(dojo|struts|session|request|application|servlet(Request|Response)|parameters|context|_memberAccess)(\.|\[\).*", ^(action|method):.*	ME, SM, SC, EE, EC
Sep 2015	\w+((\.\w+)|(\[\d+\]))|(\(\d+\))| (\['(\w|[\u4e00-\u9fa5])+'\])|(\('(\w|[\u4e00-\u9fa5])+'\)))*	[a-zA-Z0-9\.\]\[_'\s]+	**(^|\%\{)((#?)(top(\.|\['|\"\)|\[\d\]\.)?)**(dojo|struts|session|request|response|application|servlet(Request|Response|Context)|parameters|context|_memberAccess)(\.|\[\).*, ^(action|method):.*	ME, SM, SC, EE, EC

Table 1: Security mechanism evolution: 2004–2015

In January 2004 the interceptor included just one security measure: disabling Java method execution (ME) through OGNL. By default, OGNL allows Java methods to be called in a manner similar to field access. For example, attempting to set a value to `map[method()]` results in the invocation of `method` against the root object in order to get the value to be used as `map` key. Disabling the method execution is implemented using a custom OGNL method accessor and controlled using context variable `#context['xwork.MethodAccessor.denyMethodExecution']`.

4.1 Tampering with OGNL

The first vulnerability identified since the initial release relates to overwriting context variables via parameters. For example, a parameter/OGNL expression `#session['user'].role` may be used to set the role of the current user stored in the session. A more advanced expression may set a number of properties at once while still retaining original behavior, that is, setting the `newUser` parameter: `#session['user'].role=admin,#testMode=true,newUser.name'`.

In December 2004 the problem was fixed by modifying the interceptor to check if the name is acceptable, by verifying it against a black list of characters, using a condition:

```
name.indexOf('=') != -1 || name.indexOf(',') != -1 || name.indexOf('#') != -1
```

In July 2008 it was reported that the fix was incomplete, because the `#` character can be encoded using its unicode `\u0023` replacement, for example, by using a parameter `\u0023session['user'].role=admin`. This problem was fixed by adding string `\u0023` to the black list. Note that the other two already black listed characters, which could also be represented using the unicode string, were not included in the unicode form. Shortly after, the fix was further modified in a twofold way. First, the code was modified and the check routine replaced with regular expression: `[\p{Graph}&&[^,#:=]]*`. Note that the unicode replacements for the characters were no longer not excluded. In addition, the interceptor was modified to run OGNL operations against a separate temporary instance of context object (SC), without Struts-specific variables such as `session` preventing their manipulation.

At the same time, a new problem was discovered. OGNL allows accessing static fields in Java objects using the `@class@field` notation. For example, expression `@java.lang.System@exit(0).foo` can be used to call static `exit()` method, causing the JVM to exit. Static methods provided by the Java standard library can be used to perform a number of operations, including executing custom commands. This problem was fixed by adding an option to disable static method access (SM) in OGNL, and disabling it by default.

A vulnerability in this security mechanism was found in July 2010. It took advantage of the fact that context variables were still accessible through the unicode "trick" *and* that the OGNL-specific context variables controlling access to method execution, were available without restriction on the the temporary context. This allows a modification of the OGNL runtime configuration, allowing method execution and eventually custom method execution. An example sequence of OGNL expressions to perform the attack could be [4]:

```
#_memberAccess['allowStaticMethodAccess'] = true
#foo = new java.lang.Boolean("false")
#context['xwork.MethodAccessor.denyMethodExecution'] = #foo
#rt = @java.lang.Runtime@getRuntime()
#rt.exec('mkdir /tmp/PWNED')
```

Such a sequence could be encoded in a parameter, bypassing the black list, by using unicode replacements for the `#=,` characters. This vulnerability resulted in a change in the regular expression to a stricter white list of characters: `[a-zA-Z0-9\.\]\[\(\)_'\s]+`, effectively disallowing usage of the unicode replacements.

A year after this change, there was a report that while the restriction worked for regular methods it did not apply to public constructors. While execution of methods by OGNL was disabled through custom accessors, the logic did not cover constructor invocation. Some constructors may be useful to an attacker, such as `FileWriter` constructor creating or overwriting a file (`new java.io.FileWriter('filename')`). Rather than disabling constructor invocation in the existing custom accessor, the issue was fixed by disallowing a white space character, essential for constructor syntax, in the parameter name. As a result, the regular expression was modified to `[a-zA-Z0-9\.\]\[\(\)_']+`.

A few weeks later, in December 2011 the new way of bypassing the restriction was discovered [5]. It took advantage of OGNL's ability to evaluate the content of variables that already exist in the context. The attack requires setting two parameters. The first parameter uses an acceptable name but has a value containing the OGNL expression, and the second refers to the first, for instance, as array key. For example, an attacker may first set the value of existing parameter to an expression, `newUser.name=OGNL code` and then evaluate the parameter value by referencing its name, `z[(newUser.name)(0)]=0`. When the second parameter is evaluated, OGNL will attempt to establish an index of `z` property, and evaluate the expression stored in `newUser.name`. In effect, the vulnerability allows character-based restrictions on parameter names to be bypassed. This, in turn, enables access to context variables that control method execution restrictions and lead to executing custom code. The vulnerability was fixed by modifying the regular expression for acceptable name, yet again. This time, it matched characters such as `[]` or `()` only in specific context so as to disallow expressions that may evaluate other variables. An additional logic to control the expression evaluation (EE) was added to OGNL customization code.

The last vulnerability related to tampering with OGNL using parameter names was reported and fixed in August 2012. As parsing OGNL parameters requires significant processing effort it is attractive as a target for denial of service attacks. Requests with particularly long/complex OGNL expressions can be used to exhaust system resources. The problem was fixed by limiting the size of parameter names to 100 characters.

4.2 Accessing properties

Another set of security problems with processing request parameters relate to the ability to access properties of the root (action) object. In OGNL, access to the properties is controlled by the method or field access modifier in Java. For example, the `newUser` property is accessible because of the public getter `getNewUser`. If the method was defined as private or protected then the access would not be possible. The relationship between the ability to access and actual method access may not be clear or always intended. It may happen that the developer's code already has an object that would perfectly suit use within an action, but it includes a public method that should not be exposed. For example, a different implementation of the example application Section 3 may use the `User` object but does not intend to allow user to set the `role` parameter. In February

2007 a configuration parameter named `excludeParams` was provided in order to allow developers to prevent access to some properties. The parameter can be set to set of regular expressions defining patterns for parameters that should be ignored by the interceptor. Initially, the parameter was set by default to `^dojo\..*` and shortly after also `^struts\..*`.

Struts uses a dependency injection software pattern. In particular, it allows action classes to acquire certain common runtime information by implementing specific interfaces such as `SessionAware` or `RequestAware`. The example application in Section 3 implements `SessionAware` interface and corresponding `setSession` method. This instructs Struts to call it with the current server's session before action execution. Note that implementing a corresponding public getter (not required by the interface) could open up a session for manipulation using request parameter, such as `session['user'].role`. If application implements such getter, it is expected to restrict access to the parameter in the configuration.

Between 2007 and 2011 various reporters pointed out that implementing a setter, as required by Struts interfaces may also allow for manipulation. While the user may not directly access session attributes due to the lack of a getter, it may override a session object provided to the action. For example, a parameter `session.user=a` results in creation of a new Map with a `user` key. The use of this vulnerability is rather limited [6], but in certain cases it may allow unintended manipulation of application internals. In April 2012 a fix was eventually implemented by including a number of common parameter names such as `session` to the `excludeParams` list. The problem was not completely solved as it only protects a few commonly used properties and the override mechanism is still available.

A more significant problem was discovered in March 2014. Every Java object contains a `getClass` method that returns a Java class for that object. The returned `Class` object contains a number of getters and setters, in particular `getClassLoader` which returns an instance of the current class loader. Access to this object allows manipulation of the application server's internal state and allows for custom code execution [7]. The first attempt to fix the vulnerability was to add `^class\..` pattern to `excludeParams`. Within a few days a number of vulnerabilities related to an incomplete fix were reported. One, was the pattern matches `class` string at the beginning of the parameter name (`^`), but the `class` property does not necessarily has to be accessed through the root. As all Java objects contain `getClass` method, the class loader manipulation can be done through any of them, for example `newUser.class.classLoader...` Another reported vulnerability related to the fact that OGNL allows specifying parameters in upper-case form, such as `Class`, which are not matched by the regular expression. An improvement was published on the Struts web page as a hot fix, including `(.*\.|^|.*|\[('|"))(c|C)lass(\.|('|")]|\[).*`. It must be noted that, while an upper-case version was considered only for `class`, but not for `session`, `request` and others that were previously excluded. Eventually, the code performing the regular expression matching was modified to ignore case, and expression was simplified.

The series of fixes related to the class attribute, resulted in number of rather ad-hoc code changes that were rationalized in December 2014. The default set of excluded patterns was moved from the configuration file directly to the utility class used for pattern matching. Two, security unrelated, patterns were kept in the configuration file. However, the code was implemented in a way that configuration parameters overwritten the default set, effectively removing all security related excluded patterns. This problem was fixed by moving the two patterns to the code and leaving the configuration empty.

The last vulnerability relates to a special variable called `top`, implemented for Struts-specific handling of OGNL, allowing access to the root object. Effectively, this variable allowed excluded patterns of parameters to be bypassed by allowing variables to be addressed in a way that does not match the regular expressions. As a result, the `top` parameter was added to the list of excluded patterns.

Finally, a more comprehensive fix was implemented. In addition to a regular expression specifying parameter names, a custom OGNL property accessor provided by Struts was modified to exclude classes (EC) by their types and package names. For example, any attempt to access an object of a type `java.lang.Class`, or any class in `javax` package (specific to J2EE objects such as session), will be rejected. This mechanism does not have the weaknesses of the string matching approach that was repeatedly bypassed, as the verification of the object type is done at OGNL accessor level, regardless of how the expression was constructed. However, the list of excluded classes and package names is rather arbitrary.

4.3 CookieInterceptor

Since June 2007, Struts includes the `CookieInterceptor` with functionality similar to `ParametersInterceptor` but applicable to HTTP cookies. The interceptor iterates over the cookies sent with the request and sets the value indicated by the OGNL expression provided by cookie name. The developer may configure the parameters/names that be processed in the interceptor configuration.

Our analysis revealed that in several instances, problems that were applicable to both interceptors were fixed only for the parameters. At the time its first release, developers were aware of OGNL tampering issues and the rudimentary protection was already implemented for parameters, as presented in Table 1. However, it took over four years and an external reporter to implement the white list of accepted characters, which similar to that for parameters.

Additionally, issues related to accessing parameters, described in Section 4.2 were not considered for cookies for quite some time. Until April 2014 there was no restriction as to what properties can be accessed with cookies and `excludeParams` configuration was applicable only to parameters. In particular, the initial fix for the critical class loader manipulation issue was also only applied to parameters. Only after the problem was explicitly reported was the problem fixed, though only for the `class` property and not for the `session`, `request`, and so forth.

5 Analysis of security control evolution

As we traced the evolution of the security control, as outlined in the previous section, we observed a number of repeating phenomena related to introduction and prevalence of vulnerabilities, and inhibitors to the proper implementation of the security control.

5.1 The dark side of the code

One challenge is the difficulty of properly understanding every aspect of an application's operation. Modern software development is built layer upon layer of components, each encapsulating lower level detail. However, security issues often relate to low level details that are not always accessible to the developer. As a result, programmers rarely understand all the operational details of the entire stack. This problem, referred to as the "dark side of the code" in [8], can be viewed as a gap between the possible operation of the application as perceived by developer and the actual operation of which the software is capable. While [8] argues, in principle, for the existence of the dark side of the code, our study observed this phenomenon occurring in a number of vulnerabilities and confirm its existence in practice.

When the Struts interceptor developers designed the initial set of forbidden characters, they did not consider their unicode alternatives that were later used to bypass the black list based security control. The OGNL library allows the use of such replacements, however there is inadequate information concerning the scope in which the characters can be used. In its coverage of String literals, the official OGNL documentation makes vague mention of escape characters. In addition, the information provided about escaping characters is done in the context of string delimiters such as `"` and `'` and could be easily interpreted as applying only to them. The same issue applies to OGNL's ability to address properties using upper case.

Similarly, that the `getClass` method, implemented by the JVM and existing in every Java object, may be used to perform an attack might have not been expected by the Struts developer. The complexity of this attack confirms that an in-depth understanding of the Java internals, as well as the class loader specific to the application server is required in order to develop an attack vector. Additionally, the developers might have not expected that access to public constructors, exploited using file overwrite attack, could be harmful. As it is a best practice in object oriented programming to not implement constructors that cause any side effects, it may be difficult to appreciate that Java standard library includes one that allows writing a file.

Report bias A dark side can also exist when it comes to both documenting and/or interpreting vulnerability reports; the extent of the security problem may not be fully appreciated in its reporting. Security vulnerabilities are often identified by security researchers who are external to the development team.

Usually the issue is reported with a detailed description of the problem, example attack vector, and so forth. Upon receiving the information about the problem, the developers may follow a detailed report in isolation, as the prescription for the vulnerability's remedy. However, often the reporter may not have a complete understanding of the application and their report may may be incomplete; or they may liit their focus to a representative example. The vulnerability however, may have broader scope than that identified by the report or there may be further attack vectors related to the same root cause of the issue.

In Struts, the sequence of fixes related to `class` property exemplifies this phenomenon. Each time, the remediation was shaped by the way the issue was reported. This is exemplified by usage of class parameter at the beginning of expression (while it can be used for any object) and failing to provide protection for `CookieInterceptor`.

Similarly, an issue related to the exposure of constructors when using OGNL was reported as a problem that led to the overwriting custom files. Although this was only one example of the attack vector, this is how the vulnerability was described in Struts official advisory, despite the problem having a broader scope. In reality, a number of other actions are possible, provided the availablity of a suitable public constructor in the class path [9].

Security metric bias During analysis the vulnerabilities were compared to the published official security advisories. We noticed that in many cases, the CVSS score did not properly represent the problem. This can be attributed to incomplete understanding of the problem when the report was published. The CVSS documentation acknowledges that the characteristics of a vulnerability can change over time; the *temporal* metrics, used to calculate the temporal score include properties such as exploitability or remediation level. However, it is the base metrics, such as confidentiality/integrity impact that often change as the problem is better understood.

For example, CVE-2008-6504 describing the "unicode trick" to bypass the black list of characters has a CVSS score of 5.0. The impact metrics for confidentiality and integrity are, None and Partial, respectively. Another occurrence of the same problem, that resulted from an incomplete fix due to adding a temporary context object, published in CVE-2010-1870 has the same score. This is, however, not consistent with the actual impact of the vulnerability. Access to context variables effectively allows execution of custom Java code, system commands, and more. It is likely that the team was not aware of the impact when the first advisory was published; in the second case, however, the official advisory points out the variables used to control method execution. The last vulnerability reported for this problem, relating to evaluating OGNL expressions using two parameters, reported as CVE-2012-0392, has correct confidentiality/integrity impact metrics of Complete and overall score of 9.3. Even though, in hindsight, it is clear that the impact of all three issues were the same, the published information is still incorrect, something that can only be revealed by detailed analysis.

Thus, CVSS values can be biased by the understanding of the problem at the time of advisory publication. Therefore, and irrespective of the objectivity of the measure, it may not be appropriate to use CVSS in a temporal context: using it to compare (in)security of an application may lead to incorrect conclusions. The extent to which this may influence the results of past studies is a subject for further investigation

5.2 Developer's blind spots

Anticipating security problems requires a cognitive effort and often is distraction from the main objective of the developer. The research [10] shows that developers often fail to correlate security problems to their workload even if they are aware of the problem in general. Oliveira's experimental hypothesis was that vulnerabilities can be blind spots in developer's heuristic-based decision-making processes: while a programmer focuses on implementing code to meet functional demands, which is cognitively demanding, they tend to assume common, but not edge, cases. Supporting the hypothesis, the study [10] found that 53% of its participants knew about a particular coding vulnerability, however they did not correlate it with an experimental programming activity assigned to them unless it was explicitly highlighted.

Our analysis confirms existence of this phenomenon in a mature product and experienced team. Even where developers are expected to be aware of the security problems (as they considered them in the past), they may fail to consider them. When the cookie interceptor was implemented, the developers were aware of possible issues related to evaluating OGNL expressions without restrictions. Some of the restrictions were already implemented for the parameters interceptor. Yet, for three years the corresponding protection was not considered for cookies. Similarly, the access to Struts-specific `top` object, that allowed bypassing the excluded parameters list was well understood by the team. The top object facilitates the extensions to OGNL provided in Struts and, as such, it is described in the documentation. However, for almost four years when various parameters were excluded for security reasons, it was not considered in the regular expressions.

Overlooking access to public constructors could also be partially attributed the problem of developers blind spots. While, at first, the developers might have not been aware of the potential security exposure it introduces, it was no longer the case after July 2010, when usage of the constructor was highlighted to the team in the context of another reported vulnerability. Also, the example exploit, included in the advisory published on the Struts website, took advantage of a `Boolean` class constructor. While invoking the constructor was not a primary objective of the attack, it was used to facilitate it. The usage of constructor was not considered when preparing the fix for the previous issue, even though it could have been easily included.

5.3 Opportunistic fix

When developing the fix for a security problem, developers may prefer an implementation that fits their existing code. While the fix related to the root cause of the problem may be more suitable and more comprehensive, developers tend to develop fixes that are more convenient to implement and that do not cause disruption to the existing code structure.

Preventing modification of context variables or executing custom code was first implemented through simple pattern matching of OGNL expression strings, rather than limiting OGNL's capability to perform these operations, which followed later. As the the reasons for the fix are unknown, our analysis of the source code from 2004 shows that, in the code structure of the time, a more comprehensive solution required major changes in a number of helper classes and, perhaps, in the OGNL itself. Over time, and in response to numerous issues a more comprehensive solution at lower level was implemented.

Similarly, preventing class loader manipulation was first implemented by adding a pattern to the list of excluded parameters that was already in place. Only after several problems with this approach, and a number incomplete regular expressions, was a more comprehensive fix implemented which involved specification of excluded classes and packages. At the time of writing, the defense against constructor execution, relies solely on the regular expression matching, specifically the lack of the white space among allowed characters. As the regular expressions were by-passed through various tricks, it may be more suitable include protection against constructor execution at the OGNL level.

Compatibility problems Sometimes, implemented fixes are sub-optimal, as a result of issues such as compatibility with older versions and existing consumer workloads. Software consumers may rely on a particular functionality that, was subsequently identified as a source of the security problems. Thus there must be a trade-off between a comprehensive fix that breaks consumer's code, or a less comprehensive fix that may be problematic.

In Struts, many of the past attacks were related to execution of static methods. This functionality is not critical, many applications take advantage of it, and this prevented the Struts team disabling it completely, as the preferred fix. Instead, the static method execution can be enabled through configuration. As a result, the property controlling method execution disablement became a frequent target for other vulnerabilities and allowed escalating any context manipulation issue to remote code execution. Complety turning off this functionality has been planned since 2014 and developers are warned that it should be considered obsolete. Similarly, the plan to remove the `top` object, rather then controlling access to it though excluded patterns was recently announced.

5.4 Counter-intuitive mechanism

Some fixes can mean that security controls in the application can become difficult to understand or counter intuitive. While an application may not, strictly

speaking, contain a vulnerability, systems using the application may introduce their own vulnerabilities, due to incorrect usage of the security controls.

The problems of developers not properly understanding the relationship between method access and property exposure was discussed in Section 4.2. Anecdotally, many application developers are not aware of the problems arising from implementing public getters/setters for sensitive objects, or are unaware of exposing them through inheriting a class that contains such methods. The Struts team also fall victim to that problem with the `class` property, which has not been considered for over 10 years.

Initially, the `excludeParams` configuration property was not implemented as a security mechanism. It was intended to make the interceptor ignore some of the URL parameters that other layers of the application, such as JavaScript framework dojo, may use for its own purposes (for example `dojo.preventCache`). Such parameters will not have the matching properties in the action classes, and attempting to set them results in errors/exceptions, hence they are easy to spot and include in the configuration. Some patterns (such as `^dojo\..`) were set in the default struts configuration file shipped within Struts JAR file. At that time, it was expected that developers would extend this configuration in their application configuration to add any application-specific parameters. The fact that by setting their set of specific patterns, the developer would overwrite the default pattern was not a concern, as developers were aware of what non-action parameters their application uses and will include a full list.

Then, gradually, security related parameters were added to the list in order to remediate reported vulnerabilities. In order to maintain security, the developer has to find the current set of security related patterns from the default configuration file and include it when specifying, application-specific patterns. In addition, each time the application upgrades to new version of Struts, the process has to be repeated as the new version may contain new patterns. At one point the Struts team itself accidentally became a victim of this process. In version 2.3.20, released in December 2014, the code responsible for applying the patterns has been modified. Most of the patterns had been moved to a separate class handling pattern matching for both interceptor. The two remaining patterns were kept in the default configuration file. This change overwrote the patterns for security-related properties, such as `class`, by those in the configuration file. As a result, the release 2.3.20 shipped with (effectively) no security related excluded patterns and re-enabling all previously fixed attack vectors. The eventual fix to the problem was moving the two patterns from the configuration to the class itself. The configuration, empty by default, can still override the patterns if set by the application. Now, in order to include their own excluded patterns, the developer has to obtain current version of the default patterns from the Struts Java code and append their own patterns.

Assumptions about consumers A factor that contributes to the implementation of a counter-intuitive security mechanism is incorrect assumptions about the consumers' understanding of security mechanisms. The developers may not

be aware that a typical consumer does not understand all subtleties of the security framework. In analyzing the discussion on the Struts issue tracking system, we noticed that some of the initial reports on security problems were dismissed, due to a technical ability to counter default insecure behavior by specific configuration or customization. Some of these issues were eventually admitted as vulnerabilities and the default behavior was changed.

Struts developers might have not realized how counter intuitive the management of the `excludeParams` property was until it impacted on themselves. In fact, at the time of publishing the fix for the accidental overwriting (described in the previous section), one of developers opened an issue in the Struts tracking system to change the behavior and make the patterns additive.

Another example of this problem is the exposure of J2EE objects such as session objects or requests through public getters and setters required by dependency injection mechanism, as described in Section 4.2. It was assumed that application developers would implement their own protection, such as custom parameter checks. However, it is unlikely that a casual Struts consumer will be aware of such an option or the need to use it. Many application developers are not even aware that implementing a getter to match a setter is required by the interface; this is common Java practice (but clearly poor Struts practice) which can expose sensitive properties such as `session`. At the time of writing, a query [7] to a popular github repository shows 5,237 instances where a class implementing `SessionAware` interface also implements a public `getSession` method. Eventually, access to these objects was recently disabled at OGNL level, regardless of getter/setter access modifiers or their inclusion in the excluded patterns.

5.5 Evolution of phenonema

During our analysis we noticed that the above-mentioned phenonema tend to appear in the order given in Figure 3: they evolve as the developer's knowledge about the security problem and understanding of the issue and the consequences of fix increases. At first, the developer may not be aware, or only partially aware,

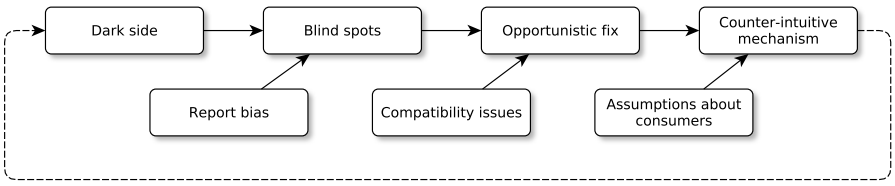

Fig. 3: Phenomena lifecycle

of a potential security problem. This may be caused by incomplete understanding of the full operation of the application or relying on incomplete advisory by the third-party. As they become more aware, they may fail to remediate the problem fully due to blind spots. The fix may be applied only for some scenarios or in some

parts of the system. Later, the develop fix may not be comprehensive, or fix the root cause of the problem. This may be a result of the the attempt to implement a fix with the least possible effort or due to the technical constraints such as compatibility with previous versions. Finally, the resulting security mechanism may be counter-intuitive resulting in incorrect use by the consumers. Note, that the end of the sequence at one level of abstraction may become a starting point for the security problem at higher level, and the counter intuitive mechanism of a framework or the library contributes to the problem with comprehending system's low level details (dark side of the code) of the consuming application.

6 Conclusion

A systematic analysis of the Struts interceptor controls was carried out over a 12 year period. A number of phenomena emerged in the evolution of the control, and these provide insights into why insufficient controls were implemented. In addition we observed that the phenomena have their own lifecycle as developers' understanding of security issues increase. Whether this can be useful in improving security processes is a topic of future research.

References

1. Massacci, F., Neuhaus, S., Nguyen, V.H.: After-life vulnerabilities: A study on firefox evolution, its vulnerabilities, and fixes. In: Proceedings of the Third International Conference on Engineering Secure Software and Systems. ESSoS'11, Berlin, Heidelberg, Springer-Verlag (2011) 195–208
2. Mitropoulos, D., Karakoidas, V., Louridas, P., Gousios, G., Spinellis, D.: Dismal code: Studying the evolution of security bugs. In: Proceedings of the LASER 2013 (LASER 2013), Arlington, VA, USENIX (2013) 37–48
3. Davidson, D.: Ognl language guide (2004)
4. Kydyraliev, M.: CVE-2010-1870: Struts2/XWork remote command execution. o0o Security Team blog (2010) online; accessed 2016-01-21, `http://blog.o0o.nu/2010/07/cve-2010-1870-struts2xwork-remote.html`.
5. Kydyraliev, M.: CVE-2011-3923: Yet another Struts2 Remote Code Execution. o0o Security Team blog (2011) online; accessed 2016-01-21, `http://blog.o0o.nu/2012/01/cve-2011-3923-yet-another-struts2.html`.
6. Long, J.: Struts 2 Session Tampering via SessionAware/RequestAware WW-3631. Code Secure blog (2011) online; accessed 2016-01-21, `http://codesecure.blogspot.ca/2011/12/struts-2-session-tampering-via.html`.
7. Ashraf, Z.: Analysis of recent struts vulnerabilities in parameters and cookie interceptors, their impact and exploitation. IBM Security Intelligence portal (2014) online; accessed 2016-01-21.
8. Pieczul, O., Foley, S.: The dark side of the code. In: Security Protocols XXIII. Volume 9379 of LNCS., Springer-Verlag (2015)
9. Dahse, J.: Multiple vulnerabilities in Apache Struts2 and property oriented programming with Java (2011) online; accessed 2016-01-21.
10. Oliveira, D., et al.: It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In: Proceedings Annual Computer Security Applications Conference. (2014)