Runtime detection of zero-day vulnerability exploits in contemporary software systems

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

[1] Ireland Lab, IBM, Dublin, Ireland.
[2] Department of Computer Science, University College Cork, Ireland.
olgierdp@ie.ibm.com, s.foley@cs.ucc.ie

Abstract. It is argued that runtime verification techniques can be used to identify unknown application security vulnerabilities that are a consequence of unexpected execution paths in software. A methodology is proposed that can be used to build a model of expected application execution paths during the software development cycle. This model is used at runtime to detect exploitation of unknown security vulnerabilities using anomaly detection style techniques. The approach is evaluated by considering its effectiveness in identifying 19 vulnerabilities across 26 versions of Apache Struts over a 5 year period.

1 Introduction

Contemporary software is routinely constructed from a myriad of components and frameworks. With the emphasis on rapid construction and reuse comes a tacit acceptance that it is neither practical nor expected that a programmer should fully understand the minutiae of every component included with their application. As a consequence, its not unusual for the programmer to focus on understanding their application at the level of the business logic, while hoping that abstraction neatly deals with the complex interoperation of its underlying components. However, these underlying components often require particular configuration and usage patterns in order to operate securely. These component requirements may be ignored by the programmer, through ignorance or as a result of coding error in the application. A programmer, while focused on using an external software component to implement one task may overlook that the application is implicitly enabled to perform another, unexpected task. Flaws can also occur within the components themselves, for similar reasons; the component developer may not have anticipated all possible use-cases. A consequence of what can be described as "dark code" [11] or "the dark side of the code" [16], is that while the programmer expects certain program execution paths, other, unexpected paths may be possible and give rise to a security vulnerability.

The software industry's approach to software vulnerabilities includes security quality assurance processes such as code reviews, static analysis and penetration testing. However, it is rarely possible to cover the subtleties of all of the numerous components and their inter-operation. In addition, during application

development, third-party components are not routinely reviewed or tested for vulnerabilities. Some of the security vulnerabilities that gained notoriety in recent years, such as Heartbleed and Shellshock, were deployed for many years before they were discovered, despite being used by large number of consumers.

Our position is that many software vulnerabilities can be attributed to programming errors that enable unexpected software behavior. Runtime verification [5] of software execution against a specified expected behavior can help identify unexpected behavior in the software. For small applications, limited requirements on expected behavior can be specified a priori, for example as a temporal proposition [5]; however, this does not scale when the requirement is to constrain emergent behavior across large complex systems of interoperating components. An alternative strategy, used by anomaly detection techniques [4, 6, 7, 17], is to learn a behavioral reference profile from system logs of past/normal behavior and use this profile at runtime to check behavior. However, practical application of this approach has tended to generate profiles with limited expressiveness [18] for relatively small and uncomplicated software components [14], such as sendmail or lpd, observed though system calls [7]. The practical challenges of applying these techniques to contemporary enterprise software, such as dealing with scale and alignment with software release processes have not been considered.

The contributions of the paper are as follows. The anomaly-detection model proposed in [15] is extended to incorporate the notion of scope which provides an effective way of dealing with scale in contemporary application systems. Based on this model, an implementation framework is developed whereby existing software testing techniques are adapted to generate profiles of expected behavior that can be used for runtime verification. The approach has been evaluated by considering its effectiveness in identifying code vulnerabilities across the 26 versions of Apache Struts over the past 5 years. Of significance is the result that anomaly-detection/run-time verification techniques would have been quite effective in identifying exploits of these zero-day vulnerabilities, prior to their discovery. To the best of our knowledge this is the first successful construction of profiles used to detect vulnerabilities in large-scale enterprise software.

The paper is organized as follows. Section 2 uses an example of contemporary software development—a microblog application implemented with Struts—to illustrate how easy it is for the programmer to unwittingly introduce a programming flaw/security vulnerability. Section 3 reviews and considers the challenges of applying techniques for anomaly detection to contemporary software systems and describes our proposed approach. Section 4 discusses the evaluation of the approach and Section 5 provides the conclusion.

2 Contemporary software development

Contemporary software is implemented using a variety of high-level programming languages, software frameworks and third party components. While application-level code may appear straightforward, enabling rapid and low-cost application development, its underlying system is a complex arrangement of inter-dependent

```
class PostAction extends ApiAction {
    private String message;
    public void setMessage(String msg) {
        this.message = msg;
    }
    public void execute(){
        String userId = getUser().getId();
        datastore.add(userId, message);
    }
    public User getUser() {
        return getSession().get(USER);
    }
}
```

```
StrutsActionProxy: PostAction.<init>
ServletInterceptor: ApiAction.setSession()
ParametersInterceptor: PostAction.setMessage("Hello")
DeprecationInterceptor: PostAction.execute()
PostAction: PostAction.getUser()
PostAction: ApiAction.getSession()
PostAction: User.getId()
PostAction: Datastore.add("lucy", "Hello")
```

Fig. 1. Micro-blog application code and trace

software whose behavior can be difficult for a programmer to fully comprehend. There are a great many examples whereby misunderstanding of contemporary software systems leads to errors in the application code that can be exploited as security vulnerabilities. We consider this problem using a simple example that runs through the paper.

2.1 A micro-blogging application

A micro-blogging social networking application provides an online facility for users to post short text messages and share them with others. The application provides a web interface and REST API for integration with various types of clients, including desktop browsers and mobile devices. The application is built using Apache Struts, a popular Model-View-Controller framework for J2EE. Struts abstracts application logic from HTTP request processing flow by encapsulating it into objects called *actions*. For example, Figure 1 shows how the action responsible for posting a new micro-blog message is implemented in just a few lines of code. Actions can be mapped to specific URL paths, such as `/api/post`. Struts can separately handle routine tasks, including parameter validation, authentication, session handling and CSRF protection, before passing parameter values to the action as associated by their setters. This facilitates the separation of business logic from HTTP processing concerns. Thus, rather than dealing with low-level operations such as accessing parameters by name, the developer need only implement a public setter. This makes the code re-usable, easier to maintain, document and test.

Application behavior can be traced as a sequence of the underlying Java method calls. For example, invoking the application using `/api/post?message=Hello` results in the included trace fragment. Each method includes a reference to its calling class. In the above, Struts creates an instance of `PostAction`, which is provided with its session and parameter data and is in turn executed, and the return value sent as response.

2.2 An unexpected vulnerability

While the implementation of the micro-blog post action appears straightforward, it can be easy for a developer to overlook subtleties in the interoperation between

the application code and the underlying Struts framework. The code in Figure 1 contains one such oversight, that results from how the `getUser` method operates with Struts, and results in a security vulnerability. Suppose that this method was not intended to be a part of the action's interface, rather, it is implemented by the programmer as a means to provide convenient access to the user object from the session. The method returns a `User` object, which is a container for various user attributes, such as name and identifier and accessor methods. The combination of a public getter for the `User` object and public setters for the object makes it possible to manipulate user information request parameters. For example, an attacker Alice can invoke `/api/post?message=Hello&user.id=frank` in order to modify the user-id stored in the session to `Frank` and submit a message on Frank's behalf. A fragment of the execution trace in this case is:

```
StrutsActionProxy: PostAction.<init>
ServletConfigInterceptor: ApiAction.setSession()
ParametersInterceptor: PostAction.setMessage("Hello")
ParametersInterceptor: PostAction.getUser()
PostAction: ApiAction.getSession()
ParametersInterceptor: User.setId("frank")
DeprecationInterceptor: PostAction.execute()
PostAction: PostAction.getUser()
PostAction: ApiAction.getSession()
PostAction: User.getId()
PostAction: Datastore.add("frank", "Hello")
```

The Struts `ParametersInterceptor` parses and interprets the `user.id` attribute as an instruction to set the `id` property of the `user` property. This results in a sequence of operations equivalent to `getUser().setId("frank")`. This oversight by the developer illustrates how easy it can be to program an unexpected execution path that compromises security. Accessing object fields though a chain of getters and setters is a useful and documented feature of Struts. However, it can come as a surprise to a programmer, who is focused more on application-level development, that something as intuitive as implementing a getter for the current user results in a security vulnerability.

This problem is further highlighted by an almost identical vulnerability (CVE-2014-0094) that was found within Struts itself. Even though the consequences of exposing a public getter may have been clear to the Struts developers, it is easy to overlook the fact that every Java object (and therefore, every Struts action) also contains a `getClass` method. This results in an unintended exposure of an action's Class object, accessed through request parameters [2]. Unfortunately, even after the problem was discovered, the implemented remedy was incomplete. The initial remedy black-listed the `class` parameter, however, it did not consider uppercase parameters (such as `Class`). Eventually, three more vulnerabilities were reported on incomplete remedies before the issue was believed addressed.

The micro-blogging application is one example of how programming error can result in a security vulnerability whose identification and prevention require an in-depth understanding of the underlying systems. However, given the complexity of contemporary systems, our position is that there will always be some aspect of the behavior that the programmer does not fully understand. For example, a study of developers by Oliveira et. al. [13] found that 53% of its participants knew about a particular coding vulnerability, however they did not correlate it with their own programming activity unless it was explicitly highlighted.

3 Security vulnerabilities and anomaly detection

The use of anomaly detection in software execution has tended to focus on relatively small-scale homogeneous applications. However, the microblogging example illustrates that even very simple contemporary applications are in fact large systems of interconnected components. This section considers the key challenges encountered in applying anomaly detection techniques to contemporary application software. For reasons of space, but without loss of generality, the discussion is focussed on a common enterprise scenario of a web application/service built on a high-level software platform (Java) and an MVC framework (Struts).

3.1 Abstraction and scope

The activity of an application is observed as a sequence of events, such as a system log. These observations can be made at different levels of abstraction. For example, the micro-blogging application activity could be observed as a series of high-level user actions, such as posting or viewing a message. At a lower level of abstraction the actions can be observed as HTTP calls to the application server. Further lower levels describe the activity of the application code and its libraries, Java virtual machine, operating system, and so forth. The objective is to use the observed activity/log of the system to build a behavior reference model for use in anomaly detection. The challenge is determining a level of abstraction that enables anomalous execution paths to identify security vulnerabilities.

Security vulnerabilities occur at different levels of abstraction. Building reference models from observations of low-level interactions, such as operating system calls [4, 7, 12] or Javascript [17] calls has been shown to be quite effective in detecting specific exploits on the application, such as buffer overflow vulnerabilities or cross-site scripting. However, the anomalous execution path in the micro-blog example cannot be detected at a level of abstraction comparable with [4,7,12,17] as observing the sequence of calls to the database looks the same as valid behavior. While one may think of this anomaly as occurring at a higher level of abstraction, making observations based on HTTP requests is not sufficient in this case (although we note that there are other anomalies that can be characterized at this high-level). An analysis algorithm is proposed [15] that discovers a level of event abstraction that provides good anomaly-detection accuracy in transaction-like behaviors; it has proven quite effective in discriminating actions that remain static from the parameters that can change within a transaction.

In our experiments we found that observing (Java) application behavior in terms of its method calls and permission checks is the most effective in distinguishing the un-expected execution paths that lead to software vulnerabilities. However, in practice it is unrealistic to base anomaly detection on every underlying method call invoked as a consequence of executing an application. For example, the few lines of the microblog application Java code in Figure 1 generate tens of thousands of method calls. Therefore, the *scope* of observation is restricted to methods that belong to the specific component(s) of interest. Section 2 considers detection of vulnerabilities in the microblog application code

and, therefore, restricts the observed scope to just those methods that belong to this application's Java package. In the experiment described in Section 4 the focus is on detecting vulnerabilities in the code that implements Apache Struts and therefore the scope is the 2400 methods defined within Struts packages.

3.2 Generating baseline activity

A baseline of observed activity must be obtained in order to build the behavior reference model for the software component. In order to be applicable to an enterprise environment, the method of establishing baseline activity has to be systematic, repeatable and aligned with application development/deployment lifecycle. Most importantly, the baseline generation must provide sufficient coverage for the normal/expected behavior of the application.

A common position is that a satisfactory baseline can be obtained as a recording of "normal" activity obtained though monitoring application operation in its target environment [6,7,17]. However this has practical limitations for enterprise software; in particular, the system must run over a period of time in order to generate a baseline with sufficient coverage. As the software industry adopts faster and more automated delivery approaches and software releases are as frequent as on daily basis, building a (normal) baseline based on observation of application in production is impractical. It would not only require the application to be active for a significant length of time without anomaly detection, it also may be impossible to obtain a sufficient baseline before it becomes obsolete.

We argue that is more appropriate to obtain the baseline and corresponding behavioral model during the software testing phase. In [16] it is suggested that unit tests might be used to generate the necessary system logs. Such a baseline is useful if the required scope and level of abstraction is consistent with the unit tests. Notwithstanding coverage, unit tests are typically constructed for a specific component, in isolation, and may only be suitable for relatively simple components or utility libraries An alternative approach is using functional and/or integration tests to also generate the baseline. Enterprise software is often subjected to extensive testing, usually highly automated and with controlled coverage. Traces resulting from such software tests may provide a reliable and systematic baseline model. However, as with unit tests, the functional test coverage is usually built only to a certain satisfactory level but is rarely complete. Additional coverage may be obtained using fuzz testing or application scanners that automatically exercise application in order to perform non-functional types of testing such as security or performance. The application scanners often use functional test execution as its starting point and then extend the coverage by exploring the application further in automated fashion.

In our experiments we found that using an application scanner allows the application system to be explored in a comprehensive, un-biased and repeatable manner and was the most effective for our task. The application scanner explores the application extensively to discover its structure, parameters, cookies and so forth, and also engages in a series of tests in an attempt to discover security vulnerabilities. While the scanner interacts via high-level requests to the

application's external interfaces, a system log/trace of the consequent low-level actions is generated. If the scanner does not discover any security vulnerabilities then it is this log that provides the baseline of expected activity.

3.3 Behavioral reference model

A number of approaches can be used to infer a reference model of system behavior based on observed past activity, with varying complexity and precision. Statistical models [8] probe various system characteristics periodically to establish a baseline of metrics. Such models, while useful for simple intrusion detection, are not sufficiently rich to detect anomalous software execution. Sequence-based techniques have been designed to infer acceptable behavior/policies for anomaly detection [7, 15] and process mining [1]. Strengths and weaknesses of existing techniques have been studied [3, 18] and expected properties of the reference model discussed [6, 14].

In many applications, activity may be contained in finite and separate units of work, where a sequence of operations that have clear beginning and end and, in general, follow some repeatable pattern is performed. Application behavior may contain multiple such transaction-like work units. For example, observing the micro-blogging application at a method call level, can be modeled as a collection of different kinds of transaction behaviors, such as posting a message or reading a message. Activity observations related to invocations of the same transaction are similar and activity related to invocations of different transactions substantially different. For this type of activity, rather than merging all transactions into a single set of short-range correlations [7], the reference model should be able to express units of activity separately from each other [15], thus at increased precision and reduced chance for mimicry attack [19].

Correlations between sequences of operations and their target values can be used to discover repeating patterns of behavior and [15] uses this to develop an automated technique to partition a system trace into a collection of behavioral norms. In our experiments we use these *behavioral norms* [15] to provide a reference model for anomaly detection. A collection of behavioral norms is generated, from baseline activity log of Java events, for a given scope and level of abstraction, each corresponding to a sequence of method calls parameterized by common target attributes. The approach is not unlike process mining [1] or sequence call monitoring [6], however, behavioral norms can be used to provide a more precise model for system-level behavior.

3.4 Runtime verification and Application Integration

While traditional logging may be useful for retrospective detection of anomalous behavior, integrating runtime verification/anomaly detection with the application provides the ability to interrupt application execution before the anomalous operations are executed. Contemporary software platforms provide integration techniques such as Aspects that may be suitable for easy and non-disruptive enablement of execution monitoring and runtime verification.

The trapping and monitoring of application execution is implemented using Runtime Verifier, a customized Java Security Manager. Its purpose is to intercept permission check requests, such as the permission to open a file, connect to server or execute a process. According to its mode of operation, this manager may log the events, check that event execution is compliant with a norm-model provided, and if required, prevent the execution of anomalous event sequences.

While the Java Security Manager is a natural integration point for monitoring and controlling application execution, it is limited to those methods that have explicit permission checks such as input/output operations or security related activity such as access to cryptographic keys. An arbitrary off-the-shelf component might not have been programmed with its own permissions. In this case it is insufficient to rely on the Security Manager, and therefore, a Java Aspect was developed that can intercept specific method calls that match programmer-specified pattern. For example, in the case of the micro-blog application, the aspect was configured to intercept method calls that are provided by classes of the application. The aspect, on intercepting a method call, invokes the Security Manager, which in turn forwards the call details as an event to the Runtime Verifier runtime. This is depicted in Figure 2.

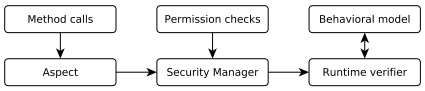

Fig. 2. Java Runtime Verifier

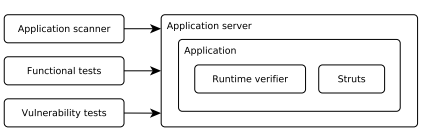

Fig. 3. Experiment setup

Using a security manager in the implementation of the Runtime Verifier permits control over application execution. Permissions and identified method calls are checked prior to execution, and thus can be validated for compliance with the behavioral model at runtime, and prevented, if considered anomalous.

4 Experimental evaluation

The previous section outlined our implementation of Java application-level anomaly detection based on behavioral norms: checking that the actual execution of an application is compliant with a model of expected behavior that was generated during software development. Our thesis is that unknown security vulnerabilities in software components can be identified as runtime anomalies arising from unexpected execution paths.

Testing this thesis using a catalog of vulnerabilities hand-crafted for the purpose, may provide insight, but their design can be contrived/cherry-picked and is not an effective evaluation of whether the approach would work 'in the wild'. We therefore decided to test the thesis against an application that used

a well-established and popular enterprise-scale software component that has a history of security vulnerabilities. In particular, the objective is to test whether vulnerabilities reported against earlier versions of software can be identified as anomalies, while those same anomalies are not reported against later versions of the software in which the corresponding vulnerability has been remedied.

In the following we considered the vulnerability history of 26 versions of Apache Struts over a five year period, starting with version 2.3.1, released in December 2011, to version 2.3.24.1, released in September 2015.

4.1 Experiment setup

Figure 3 outlines the key elements of the experimental setup. In order to evaluate Struts behavior in its typical environment, we developed a small Struts-based web application based on the micro-blogging program described in Section 2. The application makes conventional use of Struts, with a standard configuration including default interceptor stack and properties.

The application system is built automatically and the experiment is carried out separately for each version of Struts. Experiment characteristics, such as execution times and sizes, were comparable for the different versions of Struts. Experiments were orchestrated by Apache Maven and each iteration for a different Struts version comprised of two phases. In the first phase, a trace of the application system's execution is generated and from which the behavioral model is built for the given version of Struts. In the second phase the effectiveness of anomaly checking based on the generated behavioral model is verified.

4.2 Building behavioral models

A commercial application security scanner was run against the micro-blogging web application. The scanner configuration was standard and not tailored in any particular way for Struts. The same scanner configuration was repeatedly used against each deployment of the application with a different version of Struts. In each experiment, the scanner interacts with the micro-blogging web application, black-box testing an extensive collection of known vulnerabilities and misconfigurations. The Runtime Verifier was deployed with the application and used to build/check the behavior models in each experiment.

In our experiment, we consider the behavior of Struts, in terms of how it is used by the application. Therefore, the Runtime Verifier (Section 3.3) was configured to intercept all method calls within the Struts packages scope, that is `org.apache.struts2.*`., corresponding to 460 distinct methods that are used in the context of the micro-blogging web application. A single scan experiment resulted in 8963 HTTP requests to the micro-blogging web application URLs with a range of inputs, taking 8 minutes to complete. In monitoring the execution of the application during this scan, the Runtime Verifier generates a 237 megabyte baseline trace containing 2.76 million Struts events (in scope), which is analyzed, and a set of behavioral norms is generated within approximately a further 5 seconds, running on a mid-range computer. The norms are sequences of method calls,

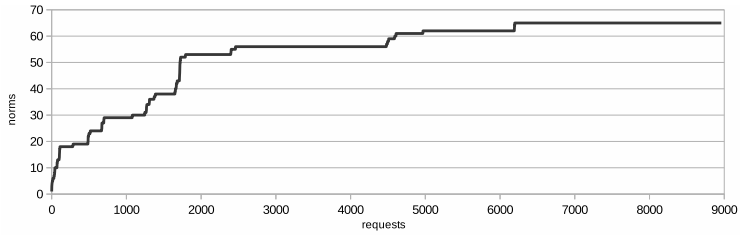

Fig. 4. Growth in behavioral norms (transactional patterns).

such as ⟨..., `UrlHelper.encode, UrlRenderer UrlProvider.getAnchor, UrlRenderer UrlProvider.isPutInContext, ...`⟩, represented as tri-grams.

Figure 4 plots an example of the number of distinct norms (transactional behavior patterns) generated, against the number of HTTP requests made by the scanner to the application deployed with Struts version 2.3.1. ¿From the start of the scan, as the number of requests increase, the number of norms resulting from the requests rapidly increase initially, and then appear to stabilize after a period. This graph suggests that our scan size of 8963 requests is adequate, after which no new norms are identified. Similar results were achieved for the other versions of Struts. Having generated a behavioral norm model for a given version of Struts and included it with the Runtime Verifier in the application deployment, the second phase of the experiment involves testing the effectiveness of using the model to detect vulnerabilities for that version of Struts.

4.3 Vulnerability tests

At the time of writing and based on the Common Vulnerabilities and Exposures (CVE) entries in the National Vulnerability Database, there are 19 vulnerabilities known to the general public for the 26 versions of Struts under study. For each vulnerability, the CVE advisory was studied alongside the vulnerable Struts code and the remediated version of the code, and an attack vector exploiting the vulnerability was developed. Of the 19 vulnerabilities, attacks for the 18 listed in Table 1 were developed; we could not find enough information to reproduce the vulnerability identified in CVE-2012-4386. For each vulnerability, we implemented an automated test case which attempts to exploit the vulnerability and verify that the exploitation was successful.

For example, CVE-2013-2115 is a vulnerability that allows a remote attacker to execute arbitrary OGNL code via a crafted request. It affects Struts JSP tags for rendering URLs. Using the tag is convenient and relieves the developer from having to manually map actions to URLs and passing parameters, thus further separating application logic from low-level details. In order to render a URL for a search action, including the current page's parameters, the developer can use: `<s:url action="SearchAction" includeParams="all">`. The tag is evaluated to `/api/search?name=Frank`. However, the code for processing the tag suffers from a security vulnerability. An attacker may add a request parameter by

including OGNL code and that code will be evaluated when processing the tag. For example, the official security advisory for this vulnerability (CVE-2013-2115) describes that an attacker may append `x=${@java.lang.Runtime@getRuntime().exec('cmd')}` to the page parameters. The OGNL code, enclosed in `${}` is evaluated and custom Java code is executed by the application. However, the attack results in activity caused not only by an unexpected Struts execution path, but also by the injected code involved in creating a process and accessing a file binary. Rather than injecting malicious code that results in significantly different behavior, each test case attempts to inject benign code that simply sets a variable that can be subsequently checked to determine the success of the attack. The objective of each test is to generate the minimal behavior needed to explore the path of the attack, but it does not engage in subsequent behavior that might be easily recognized as anomalous in its own right. In this way the test case is intended to represent a worst-case scenario for anomaly detection.

In many cases developing the vulnerability test cases required some effort. A number of the vulnerabilities are not clearly described in their respective advisories and sometimes the details are intentionally undisclosed. For example, CVE-2013-4310 (discussed below in this paper) is described as allowing "to bypass security constraints under certain conditions". In order to prepare the test case for this vulnerability it was necessary to understand its nature though analysis of source changes of Struts.

Vulnerability test results. Table 1 contains[3] the outcome of testing each of the 18 reported vulnerabilities against each of the 26 versions of Struts. Each

CVE ID	2.3.1	2.3.4	2.3.14	2.3.14.1	2.3.14.2	2.3.15	2.3.15.1	2.3.16	2.3.16.1	2.3.16.2	2.3.16.3	2.3.20	2.3.24	2.3.24.1
2015-5209	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−−
2015-1831	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	+++	−−−	−−−
2014-7809	++−	++−	++−	++−	++−	++−	++−	++−	++−	++−	++−	−−−	−−−	−−−
2014-0116	+++	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−+	−−+	−−+	−−+
2014-0113	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−+	−−+	−−+	−−+	−−+
2014-0112	+++	+++	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−
2014-0094	+++	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−
2013-4316	+++	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-4310	++−	++−	++−	++−	++−	++−	++−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-2251	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-2248	+++	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-2135	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-2134	+++	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-2115	+++	+++	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-1966	+++	+++	+++	−++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2013-1965	+++	+++	+++	−++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2012-4387	+++	+++	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−
2012-0393	++−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−	−−−

vulnerable attack successful anomalies not vulnerable attack failed no anomalies

Note that some outcomes were identical for different versions of Struts, and in the interest of saving space.

Table 1. Attack Outcomes on different versions of Struts.

[3] Presented as a table, keeping in mind Edward R. Tufte's (2004) observation that *"small non-comparative highly labeled data sets usually belong in tables"*.

table cell contains three outcomes. The first indicates whether it was reported that the particular version of Struts was indeed affected (+), or not (–) by the vulnerability. The second outcome specifies whether the execution of the attack for that vulnerability was successful (+), or not (–). The third outcome, specifies whether th Runtime Verifier detected anomalous behavior during execution of the test case (+), or not (–). For example, the outcome +++ means that the version had the reported vulnerability, that the attack test case successfully executed and that anomalies were detected (true positive).

Considering CVE-2013-2115, the URL tag vulnerability described above, we see from Table 1, that the attack test-case successfully exploited this vulnerability, and was detected as an anomaly, for all versions 2.3.1 – 2.3.14.1 publicly announced to be vulnerable (+++). A closer examination of the execution trace fragment, generated from the attack test case,

```
DefaultUrlHelper DefaultUrlHelper.translateAndEncode
DefaultUrlHelper DefaultUrlHelper.translateVariable
OgnlInvoke invoke
ServletUrlRenderer ComponentUrlProvider.getAnchor
ServletUrlRenderer ComponentUrlProvider.isPutInContext
```

identifies the anomaly as OGNL code used in rendering the URL. Examining execution traces may help identify that part of the code that is responsible for the vulnerability. Indeed, a study of subsequent versions of the Struts source code that repair the vulnerability reveals that the issue was attributed to `translateVariable` that invoked OGNL processing. In the repaired code, the method was removed as unnecessary. Carrying out the same test-case on subsequent non-vulnerable versions results in an unsuccessful attack and no anomalous behavior (– – –), which is as expected. The corresponding trace fragment for those versions shows that no OGNL code execution—the root cause of the vulnerability in previous versions—was observed in the context of URL rendering:

```
DefaultUrlHelper DefaultUrlHelper.encode
ServletUrlRenderer ComponentUrlProvider.getAnchor
ServletUrlRenderer ComponentUrlProvider.isPutInContext
```

Some tests executed with unanticipated outcomes. A surprising result –++, indicates a successful attack, with the anomaly detected, for Struts version 2.3.14.1 for which no vulnerability was reported in CVE-2013-1965 and CVE-2013-1966. A closer examination of these two vulnerabilities confirms that, contrary to publicly available information, version 2.3.14.1 is indeed vulnerable.

False negatives In most cases, Table 1 reports that the anomaly detection identified successful attacks on vulnerable versions, while unsuccessful attacks on non-vulnerable versions were not identified as anomalous. The table, however, reports two vulnerabilities with a false negative result (++–), that is, a successful attack on a vulnerable version for which anomalous behavior was not observed.

One false negative arises from CVE-2014-7809: a CSRF vulnerability caused by a predictable token generated using a weak generator. It allows an attacker, knowing a previous value of a token, to predict the value of the next token and to use it to perform an attack. Although the attack is caused by a simple coding error, we argue that it does not arise from an unexpected path of code execution. A CSRF attack as a request using a token generated by the attacker results in

exactly the same behavior as a legitimate request made by the user. As such, it can not be detected as an anomaly in the execution path.

Another false negative arises from CVE-2013-4310, which reports an ability to bypass security constraints. Our investigation discovered that this vulnerability is only applicable to applications that have a somewhat unusual security mechanism. As mentioned in Section 2, actions in Struts are the basic unit of an application's business logic and are normally mapped to specific URL paths (such as `/api/post`). Struts offers an alternative addressing through URL parameters with `action` prefix, such as `/api/other?action:post`. The vulnerability describes a scenario when a security control, implemented outside struts, based on specific URL pattern is bypassed using the alternative addressing. It could be argued such scenario does not really describe a Struts vulnerability, but rather a faulty security control that documented feature of Struts allows to facilitate. In our experiment, the attack exploiting CVE-2013-4310 was undetected because the application we implemented included use of `action:` prefixes. Addressing actions though parameters was considered a normal behavior of the application and, when executed during the attack, did not cause anomalous behavior.

False positives In two instances, the test cases result in false positives, that is, anomalous behavior is detected for an unsuccessful attempt to exploit non-vulnerable version (−−+). This outcome is observed for CVE-2014-0114 and CVE-2014-0116 in versions where these vulnerabilities are repaired. They are variants of the problem discussed in Section 2 where the internal state of an application can be modified through a chain of getters and setters, and in this case, though crafted cookies. A study of the attack test-cases reveals that the anomalous behavior is related to the special treatment that Struts gives to particular cookie names. The original vulnerabilities were repaired by adding a blacklist of disallowed cookie names (such as starting with `class`). Thus, processing a normal cookie results in a behavior that is different to processing a cookie with blacklisted name. However, in its standard configuration, the application scanner does not generate a request involving a Struts black-listed cookie and, therefore, the generated model of expected behavior does not include an execution path corresponding to the security processing of a blacklisted cookie. Thus, the test-case, while not an attack, is flagged as an anomaly.

Overall, this part of the experiment indicates that all vulnerabilities that could be attributed to unintended code execution path in struts were successfully detected. It also shows that, with one exception of black-listed cookies, in non-vulnerable versions, where malicious request is properly handled by the application no anomalous behavior is reported.

4.4 Functional tests

On generating an expected model of behavior (for a given version of Struts) we check that its norms are sufficiently complete by engaging a further standard application scan and confirm that no anomalies are identified. This confirms

that 'normal' expected behavior is properly recognized by the Runtime Verifier. However, for the purposes of the evaluation, we are interested in confirming that the anomaly detection can also discriminate between attacking behavior that exploits a vulnerability versus other behavior that executes code in the region of the vulnerability, but does not actually exploit the vulnerability. To this end, a suite of functional tests were developed to check this ability to discriminate.

For example, the URL tag vulnerability CVE-2013-2115 described in Section 4.3 involves passing a crafted value through a URL parameter. The test calls an application using an additional parameter but without OGNL code. Some vulnerabilities require more advanced test cases. For example, the attack test-case for CVE-2013-2251 involves passing a crafted string through a Struts-specific request parameter, allowing indirect action addressing. We developed two further functional test-cases that check this particular functionality. First uses the indirect addressing, but with a correct action name, and checks that the requested action was called. Second has an incorrect action but checks whether the application replied with expected/corresponding error. The purpose of these tests is to check whether the anomaly detection actually reacts to a genuine vulnerable path of execution or whether the path from related valid functionality is flagged as an anomaly. By making sure that the functionality is exercised we can distinguish between these two cases, exercising particular functionality normally (even if in error scenario) and during the attack.

Overall, we developed 19 test cases that explore non-attacking behavior in the region of the 18 vulnerabilities. The outcome-score of each test is similar to the vulnerability tests and represented using two values. The first outcome value reports whether the test was successful, that is whether the tested functionality worked correctly. Note, that because we have also tested an application's response to an incorrect request, a successful outcome may mean that the application correctly responded to an incorrect value, such presentation of an error page. The second outcome value reports whether an anomaly was detected during execution of the test. Most of the test outcomes are indicating a successful test with no anomalies. However, in two cases the outcome indicates test failure with no anomalies is observed. These are the test cases for indirect action addressing using `action:` prefix. The test fails for all versions from 2.3.15.2. This is because, as a response to CVE-2013-4310, this functionality was disabled.

4.5 Experiment insights

The application scanner may trigger an unexpected behavior, which if unattended to, becomes part of the model used in runtime verification. However, this phenomenon was not observed during our experiments. Furthermore, we assume that any vulnerability identified during scanning will be remediated by modifying the source code and the scan repeated.

Anomaly detection/prevention adds a performance overhead that should be considered. In order to integrate anomaly detection with the application we used a Java Security Manager and AspectJ. These tools are routinely used for

implementing security mechanisms and their performance impact has been investigated [9,10]. During the the Struts experiment, the average time to process an HTTP request from the scanner to the application took 4950μs without instrumentation. With the Anomaly Manger enabled for runtime verification, the average time increased by 3.85% to 5140μs. However, the increase depends on how much of application activity is covered by runtime verification: in the experiment this was limited to the Struts library.

Part of our experimental setup involved inspecting the codebases of different versions of Struts in order to identify the vulnerable code and implement the attack tests. In carrying out this detailed code-level review we observed a number of programming phenomena across the different versions. In particular, the phenomena that some generic functionality of Struts allows for a specific execution scenario that compromised security. The otherwise harmless features, such as addressing actions, setting their parameters and evaluating expressions, when used in a particular way allowed unintended operations. For example, five vulnerabilities exploit the feature that allows setting action properties through HTTP parameters and accessing sensitive objects such as session or class loader.

Overall, we observed that the majority of the programming issues relate to rather simple programming errors. In particular, while a general functionality was implemented, a specific, unexpected execution pattern was not considered nor handled by the code. In some cases, such as accessing the class loader via parameter (CVE-2014-0094), we conjecture that the developers were surely aware that accessing properties though parameters can be a security risk. Some specific parameters, such as session object, were black listed but others that are less obvious, such as class, were not. In other cases, when a vulnerability was identified in one part of the framework it was not immediately correlated to another part that was also vulnerable [2]. For example, the fix for CVE-2014-0094 addressed class loader manipulation though request parameters but did not provide the fix for the same attack using cookies. Our analysis of Struts vulnerabilities would seem to confirm other studies [13] that indicate that developers tend to repeat security errors even when they are aware of particular vulnerability.

5 Conclusion

We argue that many of the security vulnerabilities that are caused by programing error can lead to unexpected execution paths that can be detected using anomaly detection techniques. However, applying such techniques to contemporary, enterprise software is not trivial and poses a set of challenges such as selecting the scope and abstraction level of software monitoring, establishing a base line behavior and choosing suitable behavioral model.

We propose a methodology for putting anomaly detection into use for contemporary software components and apply it to Apache Struts as a part of large scale application. Our experiments demonstrate that it is possible to learn a sufficiently rich model of the application's expected use of Struts such that it can be used to detect anomalies in its subsequent use of Struts. Indeed, results indicate

that all 16 execution path-related vulnerabilities identified over 26 versions of Struts over 5 years are effectively identified as anomalies. While the experiments were comprehensive, they are limited to Struts vulnerabilities. Nevertheless, we believe the results point to the potential of using anomaly detection techniques in contemporary software and that further research on this topic is worthwhile.
Acknowledgement. This work was supported, in part, by Science Foundation Ireland under grant SFI/12/RC/2289.

References

1. van der Aalst, et al.: Workflow mining: Discovering process models from event logs. Knowledge and Data Engineering, IEEE Transactions on 16(9) (2004)
2. Ashraf, Z.: Analysis of recent struts vulnerabilities in parameters and cookie interceptors, their impact and exploitation. IBM Security Intelligence portal (2014), online; accessed 2015-05-21
3. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection for discrete sequences: A survey. IEEE Trans. on Knowl. and Data Eng. 24(5) (2012)
4. Creech, G., Hu, J.: A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. IEEE Trans. Comp. (2014)
5. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. (2004)
6. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceedings of the Annual Computer Security Applications Conference (2008)
7. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: IEEE Symposium on Security and Privacy (1996)
8. Helman, P., Liepins, G.E.: Statistical foundations of audit trail analysis for the detection of computer misuse. IEEE Trans. Software Eng. 19(9) (1993)
9. Herzog, A., Shahmehri, N.: Performance of the java security manager. Computers & Security 24(3) (2005)
10. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: Proceedings of the 3rd International Conference on Aspect-oriented Software Development (2004)
11. Holzmann, G.J.: Code inflation. IEEE Software 32(2) (2015)
12. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Trans. Dependable Secur. Comput. (2010)
13. 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)
14. Patcha, A., Park, J.M.: An overview of anomaly detection techniques: Existing solutions and latest technological trends. Comput. Netw. 51(12) (2007)
15. Pieczul, O., Foley, S.: Discovering emergent norms in security logs. In: Communications and Network Security (SafeConfig), 2013 IEEE Conference on (2013)
16. Pieczul, O., Foley, S.: The dark side of the code. In: Security Protocols XXIII. LNCS, vol. 9379. Springer-Verlag (2015)
17. Raman, P.: JaSPIn: JavaScript based Anomaly Detection of Cross-site scripting attacks. Master's thesis, Carleton University (2008)
18. Tan, K.M.C., Killourhy, K.S., Maxion, R.A.: Undermining an anomaly-based intrusion detection system using common exploits. In: Proceedings of the 5th International Conference on Recent Advances in Intrusion Detection (2002)
19. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: ACM Conference on Computer and Communications Security (2002)