Journal of Network and Computer Applications 60 (2016) 95–112

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Review

Web application protection techniques: A taxonomy

Victor Prokhorenko, Kim-Kwang Raymond Choo, Helen Ashman

University of South Australia, Australia

ARTICLE INFO

Article history:
Received 16 July 2015
Received in revised form
28 October 2015
Accepted 19 November 2015
Available online 11 December 2015

Keywords:
Web IDS
Web protection
Web application
Web security

ABSTRACT

The growing popularity of web applications makes them an attractive target for malicious users. Large amounts of private data commonly processed and stored by web applications are a valuable asset for attackers, resulting in more sophisticated web-oriented attacks. Therefore, multiple web application protections have been proposed. Such protections range from narrow, vector-specific solutions used to prevent some attacks only, to generic development practices aiming to build secure software from the ground up. However, due to the diversity of the proposed protection methods, choosing one to protect an existing or a planned application becomes an issue of its own.

This paper surveys the web application protection techniques, aiming to systematise the existing approaches into a holistic big picture. First, a general background is presented to highlight the issues specific to web applications. Then, a novel classification of the protections is provided. A variety of existing protections is overviewed and systematised next, followed by a discussion of current issues and limitation inherent to the existing protection methods. Finally, the overall picture is summarised and future potentially beneficial research lines are discussed.

Contents

E-mail addresses: Victor.Prokhorenko@mymail.unisa.edu.au (V. Prokhorenko), Raymond.Choo@unisa.edu.au (K.-K. Choo), Helen.Ashman@unisa.edu.au (H. Ashman).

http://dx.doi.org/10.1016/j.jnca.2015.11.017
1084-8045/© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

This review examines recent web-based protection techniques from the perspective of a detailed classification. Perhaps, due to a large variety of web technologies used in modern web application development, the majority of existing reviews commonly cover only a specific type of attack or summarise a class of protection techniques (e.g. Asghar Sandu et al., 2011; Hossein Manshaei et al., 2013; Scholte et al., 2012; Du et al., 2011; Mitchell and Chen, 2014). Comprehensive reviews aiming to systematise currently existing techniques have been conducted as well (Li and Xue, 2014). However, given the multitude of incompatible views, this review aims to provide an alternative and more holistic classification of the existing web application protection techniques. A brief overview of security software evolution is first provided to explain the current landscape of various software protection techniques.

The history of software development shows that software applications are generally designed to provide features that appeal to consumers rather than with security in mind (the latter is also known as *security by design*). Retrofitting security in such software is usually not economically viable, due to the costs and resources required (e.g. integrating security in an application with a large code base without introducing compatibility issues). A common approach is to implement an additional layer of protection on top of the existing application, and these protection layers are known as *envelope protections*.

In the 1980s, the main security threats were viruses, thus antivirus software came into existence. With the development and penetration of networks in computing, firewalls emerged as a separate class of security software. Firewalls were designed to protect against malicious network-based activities (including virus propagation over networks). However, strict permission models employed by early firewalls were not flexible enough in some situations, as the control of the network connections happens at a lower level. For example, a traditional firewall could completely block a TCP port, but could not selectively pass only the packets with some specific data. This type of limitation caused Intrusion Detection Systems (IDSs) to be developed.

The purpose of an IDS is to serve as an intermediate layer between the application being protected and the users. The layer would analyse user actions and detect possibly malicious activity. Once a suspicious activity is detected, such a protection layer would raise an alarm to warn the system administrator. An obvious improvement is to not only detect, but also prevent (if possible) the attack, as implemented by Intrusion Prevention Systems (IPSs).

Another firewall improvement emerged due to the growth of computing power available. As computational speeds increased, a direct analysis of each byte in all the network packets became feasible for firewalls. Such technology was named Deep Packet Inspection (DPI) and is now commonly used on the market. In contrast to traditional firewalls which made decisions on whether to pass or reject a packet based on the packet headers, a DPI firewall can analyse the packet data payload as well.

In some ways, a network-level IDS can be considered to be the same thing as a DPI firewall. Both approaches require analysing all of the network packets passing through, paying attention to the packet data. The difference mainly lies in the decision phase, whereas the data capturing and analysis are technically the same.

Web technologies were born out of the necessity to communicate with remote clients. With further growth of the networks, the popularity of web technologies increased. Initially, web sites were used to serve static content, however with growing user demands, various techniques were introduced to serve dynamic content as well. Such techniques include both client- and server-side scripting. While the code executed at the client side does not directly affect the web application, introducing the notion of server-side programming brought complex web applications into existence. The increasing complexity of the server-side code caused bugs to appear in web applications making it possible to perform successful attacks.

Due to conceptual differences introduced by web technologies, traditional protection mechanisms such as antiviruses, firewalls or network level IDSs are not directly applicable to protecting web sites. For example, non-DPI firewalls would only either provide or block access to the web server as a whole. Network level IDSs would face the same kind of limitation without having additional information about the web application structure. Traditional host-side antivirus protections solve a somewhat different problem of detecting and preventing malicious code execution. Modern antiviruses tend to monitor multiple virus intrusion vectors such as network or USB disks and the strict difference between antiviruses and IDSs can thus be considered to be blurred to some extent. However, a conceptual difference in the context of web applications still exists. In case of antiviruses, obtaining a virus signature does not eliminate the threat, and the signature has still to be stored for further detection. However, having knowledge about a web application attack, signature provides enough information for the problem to be fixed. Once the vulnerability is removed, there is no more need in storing the attack signature (perhaps apart for the legacy software which is not updated any more).

A somewhat parallel branch of protection techniques includes various source code analysing approaches. Due to the requirement of having access to the source code, such approaches are applicable either during the software development stage, or for software with open source. While such a requirement makes it impractical to use such approaches for proprietary third-party applications, the spike of popularity of web technologies made it possible to apply source code analysis more frequently as interpreted languages (such as PHP) are commonly used for web development.

The limitations inherent to traditional security software in the context of specific applications rather than at the general host or network level caused the creation of application-specific protection techniques. Tailoring protection to a specific application allows fine-grained control. Such protection would, for example, be able to block access only to a subset of application functions. Applying application-specific protection for traditional desktop applications might not be meaningful in general, as both the user and the application are located on the same host. However, such protection can be used for separated client-server architecture software to protect the server side. Client-server architecture based software has become quite popular in the form of web applications motivating the need to provide security in diverse web development environments. Web-based application-specific protections are also somewhat more universal due to the open nature of protocols involved (mainly HTTP and SQL). In contrast, a protection implemented for a specific application that uses proprietary protocols cannot be deployed for other applications.

Overall, history shows that security software mainly evolved to be more targeted. This includes transition from protecting a system as a whole (antiviruses) to protecting a specific network application (traditional firewalls) and further to focus on finer granularity of specific application behaviour aspects (DPI). While a generic protection may not be possible for all applications and protocols, it is possible to implement an effective protection for a specific target (against certain types of attacks). Therefore it is more cost-effective to implement a protection for widely used universal protocols (such as HTTP) rather than implementing multiple application-specific protections. While very diverse, web applications are using a common underlying protocol. Such standardisation makes it possible to implement a generic protocol-level protection which would be applicable to large amount of deployed web applications.

The growing dependence on web applications for everyday tasks like online shopping attracts popular non-technical media attention when attacks against high-profile web applications occur (Forbes, 2013). Despite the considerable efforts put into protecting popular applications against various attacks, even the well understood types of attacks such as SQL injections commonly occur on a large scale (Ars Technica, 2015a, Wired, 2012, 2008). These kind of massive server-side vulnerabilities can be explained by the use of popular web development frameworks – if a vulnerability is in the frameworks itself, all the web applications based on that framework become vulnerable (Ars Technica, 2015a, Wired, 2008). In some cases the web browser itself can be vulnerable (Ars Technica, 2015b); however, popular weaknesses are mostly attributed to the server-side (CWE, 2011). Even significant number of government web sites is vulnerable to web-oriented attacks signifying the high importance of securing popular web application (Wired, 2011).

Therefore, this review is focused on various web application protection techniques, although some of the techniques covered (e.g. static code analysis) are generic enough to be applied to any kind of applications. The key contribution of this paper is to provide an in-depth and comprehensive web application protection technique classification.

The paper reviews 69 publications between 2000 and 2014 in sources such as ACM, IEEE, Springer and Science Direct using the following keywords: web security, web intrusion detection, web protection, intrusion detection, web defence, web attacks prevention, web IDS.

The outline of this paper is as follows. Section 2 presents a novel classification of web application protection techniques. Section 3 discusses and categorises various existing protections according to the proposed taxonomy. In Section 4, we examine the limitations of existing protection techniques. In the final section, we conclude by presenting a conceptual protection technique to address several of the limitations identified in this paper.

2. General web application protection classifications

Web application protection techniques can be classified by multiple factors. Such factors may include the type of web application being protected, the class of attacks the protection method detects or prevents, various generic traits of the approach or the subject being protected.

This section presents a novel general classification of web application protection techniques (Table 1).

The presented classification uses two independent sets of properties to categorise protection techniques. Namely "Subject of observation" and "Decision base". The "Subject of observation" property includes *"Inputs"*, "*Application*" and *"Outputs"*. "Decision base" property consists of "*Statistics*", "*Policy"* and "*Intent*". As the name suggests, the first property defines what the protection analyses. For example, a protection may monitor user input or application output, or both. The second property defines what the decisions are based on. Actual decisions include attack reporting or prevention. Decisions based on statistics (commonly referred to as anomaly detection) rely on an assumption that attacks are abnormal (rare) events. Policy-based decisions are based on the prior knowledge or wishes of the protection developer or administrator and are commonly used in rule-based approaches. In contrast to policy-based, intent-based protections are explicitly

Table 1
Web application protection techniques classification criteria (Authors' compilation).

Decision base	A. Statistics (anomalies – probabilistic view)	B. Policy (protection developer/administrator knowledge or wishes-rule-based approaches)	C. Intent (application developer intentions)
Subject of observation	1. Inputs (HTTP GET/POST parameters, Cookies, Headers, URLs requested – user behaviour)	2. Application (source code and layout-white box approaches)	3. Outputs (generated page, database/network/file system activity – application behaviour)

considering the application developer intent (usually through the application source analysis).

Two sets of properties are required to be specified to determine the class a protection technique belongs to. For example, {B 1} denotes a protection monitoring the application inputs and making decisions based on the policies defined by the protection user (traditional rule-based approaches). A more complex example would be {A 1 3} or {A 3} – typical black-box anomaly detection techniques.

Note, that while the classification presented does not explicitly contain client-related aspects, however, a brief overview (along with reasons of omission) of such aspects is present in Section 2.8. The next section explains the general conventions used in the presented classification and subsequent sections investigate this classification in detail.

2.1. Web application protections vs. development practices

While secure construction of web applications is included in some protection taxonomies as a separate category (Li and Xue, 2014), such construction techniques are, strictly speaking, not protections, but rather generic security-focused development practices and therefore are not included in the proposed classification. These security-oriented development improvements may involve using special frameworks, third-party APIs or even specialised programming languages (Batory et al., 1998; Scott and Sharp, 2002; Chong et al., 2007; Robertson and Vigna, 2009; Martin et al., 2006; Huang et al., 2008). The improved development practices focus on creating an error-free (or at least to minimise errors) applications, making such practices applicable for new web applications. However, given a functioning legacy web application and the high cost of developing a new one to replace the existing application, the choice is usually made in favour of legacy web application oriented protections.

A possible complication arises if the developer of a legacy application is not available any more, leading to difficulties in further application maintenance. To address this complication, various tools can be used to automatically extract different architecture aspects from the existing source code (Hassan and Holt, 2002; Moonen, 2001). While such tools do not directly affect application security, the results obtained may be useful to design a protection tailored to a specific application.

Although some protection approaches aim to be quite generic (Ray and Ligatti, 2012), generally protections depend on the programming language used and are mainly targeted towards the most popular languages such as PHP, ASP or Java (Merlo et al., 2007; Tripp et al., 2009). While still language-dependent, some approaches may also focus on inter-language or inter-component interaction analysis for heterogeneous environments (Furr and Foster, 2005; Grechanik et al., 2006). Overall, this higher-level classification reflects more a financial and time-related choice rather than focusing on technical differences. Furthermore, as soon as a new application comes into existence, it becomes possible to apply any of the protection techniques to protect the application.

Security testing and vulnerability scanning techniques are standing in between development practices and legacy-oriented protections, as such techniques typically test or scan (at least partially existing) applications, but still require modifying the application code to rectify the detected vulnerabilities. While automated rectifying may be possible in some cases (Huang et al., 2004), security testing and vulnerability scanning essentially lack the decision component altogether and offload the burden of decision to the application developer. Such techniques therefore fall into either {2} or {1 3} class, depending on whether white-box or black-box approach is employed. Later sections provide an in-depth overview of the technical aspects of the proposed classification.

2.2. Statistics-based (probabilistic) decisions

Protections using statistics-based decisions aim to avoid the problem of zero-day attacks by making dynamic decisions based on previously gathered statistics, rather than on predetermined list of rules. Such protections assume that an attack is an abnormal event not usually observed in every day usage of the system. Therefore, it is possible to characterise the “normal” system behaviour and detect any deviations at later time. Two general steps are required to implement such a type of protection. The first step, commonly known as the training phase, involves determining what the normal system behaviour actually is. Obtaining the normal usage behaviour might be useful by itself, for example for web site optimisation purposes (Diebold and Kaufmann, 2001), but here it is an essential step so as to have a comparison baseline to measure later activity. The second step is the actual detection process. Depending on the training type, appropriate comparison methods are used to check whether the currently observed behaviour is significantly different from the normal one recorded previously (Chandola et al., 2009). To account for future changes in normal usage, a common improvement of constantly adjusting the initial usage model is implemented. Protection methods from this category mainly differ in the learning model selected, including support vector machines, data mining or gene programming (Li et al., 2004; Kukielka and Kotulski, 2008; Han and Cho, 2006; Sung and Mukkamala, 2003; Skaruz, 2009; Lee and Stolfo, 2000; Shum and Malki, 2008).

Despite using different mathematical abstractions, such statistical approaches still have several common problems to solve. The first problem is to sustain a clean training environment; if an attack happens during the training phase, such activity can be recorded as normal behaviour (Lee et al., 2001; Park et al., 2004). This means that actual attacks occurring later will not be detected, these errors being known as *false negatives*. The second problem is the necessity to cover all usage aspects for a given system, so that rarely-used functionality is not mistakenly detected as a malicious activity at later stages. Given that some actions might be rare in their nature, such actions can end up missing from the recorded normal behaviour profile and be wrongly classed as attacks at later stages, these errors being known as *false positives*. Furthermore, if a web application is changed, the typical usage of the application may change significantly. Thus, some approaches specifically address the usage drift caused by application changes (Maggi et al., 2009). The third problem lies in the correct choice of factors to consider while building the normal usage model. Attack detection accuracy highly depends on this choice. For example, if only character distribution in network requests is considered as a factor to observe, but not the length of the packet, an attacker can then encode the attacking string in a way to match the normal character distribution at a cost of making the request packet longer (Mason et al., 2009). The protection would not trigger an alarm, as the observed factor (character distribution) is not different from the normal usage, whereas considering the average request length as well would make it possible to detect the attack (if the usual request length is significantly shorter). Finally, while no evidence exists yet, the fourth problem is the theoretical danger of malicious re-training of the system (Bai and Kobayashi, 2003), which exploits the gradual ongoing adjustment of the ‘normal’ behaviour model. This problem is only applicable to protection systems which do adjust the normal usage model during run time; however, it is still important as continuous self-training is a common technique.

The detection phase very much depends on the normal usage model, but still has a common problem of choosing optimal comparison thresholds (e.g. assigning appropriate weights to all the factors considered). Essentially, the protection system must be configured to know "how different" the detected action should be from the normal behaviour to be reported as an attack. While extensively researched, the anomaly-based approaches are criticised because of fundamental differences between the security domain and other areas where machine-learning is used. These differences include the lack of available training data in security domain and the fact that machine-learning strength lies in detecting activities similar to those previously seen (rather than detecting dissimilar, unknown ones) (Sommers and Paxson, 2010).

2.3. Policy-based decisions

In some cases the decisions made by a protection can be controlled by rules based on prior knowledge or wishes of the protection developer or administrator. Such rules may not necessarily depend on the specific application being protected (for example, source- or time-based policies). In such policy-based protections, the decision of what is considered to be an attack is made in advance by the developer or a user of the protection system. During regular usage, the protection is then continuously monitoring the system and running the observations through the rule set to make decisions. Rule sets describe various attack traits such as a byte signature present in the network packets. While such approaches are precise by design, the actual detection results highly depend on the rule set deployed. Keeping the rule set up-to-date requires constant attention and effort as new attacks continue to emerge. An inherent downside of such approaches is the problem of zero-day attacks – the protection is unable to detect unknown attacks. Another challenge of rule-based approaches is the possibility of bypassing the protection by obfuscating the attack signature in the case when the attacker knows the exact rule set deployed (Schmitt and Schinzel, 2012).

Attack detection precision for rule-based approaches can be improved with the use of comprehensive rule set. The main two lines of improvements are to enhance the rule set and to extend the rule definitions. However, regardless of rule number and detail level, the policy-based protections are conceptually limited by their reactive nature.

2.4. Intent-based decisions

In contrast to for policy- and statistics-based decisions, intent-based decisions consider the original intentions of the application developer. Ideally, the developer intentions should be in the form of formal specifications covering some aspects of the application behaviour (such as data flow definitions or database access workflow). In practice, however, such formal documentation is rarely available. The next best source of application developer intentions is the application source code (especially if the original developer is unreachable).

Similar to statistics-based decisions, there are two steps involved in the intent-based decisions. First, various intentions are reconstructed from the application source (learning phase). For example, such intentions may include the structure of SQL statements issued by the application (Halfond and Orso, 2006). The extracted intentions essentially form expectation of how should the application behave under normal operating conditions. During the second step the execution environment is monitored and alarm is raised if some unexpected situations occur. Depending on the type of the intentions extracted, such unexpected situations could, for example, include an unexpectedly structured SQL statement issued to the database or unexpected number of command-line arguments passed to an external application.

In summary, protection techniques making decisions based on application developer intentions effectively try to verify whether the exhibited application behaviour is similar to what the developer intended the application to do (at least in some aspects).

2.5. Inputs analysis

A significant class of attacks, such as SQL injections or XSS attacks, is made possible because of incorrect handling of user-supplied data. Thus, a whole category of protection methods focused on user input monitoring and sanitising has emerged (Haldar et al., 2005; Livshits and Chong, 2013; Weinberger et al., 2011b; Balzarotti et al., 2008; Saxena et al., 2011; Samuel et al., 2011; Hooimeijer et al., 2011; Wassermann and Su, 2008). Regardless of the exact technical steps involved, the general idea behind these protections involves continuous user input monitoring, and possibly alteration, to conform to a predefined set of rules (or previously collected statistics). The user inputs (commonly HTTP GET/POST parameters, cookies or headers) are monitored and analysed at the moment they arrive to the application being protected. Depending on the implementation, in case a non-conforming input is detected, either the whole transaction is dropped, or a recovery (sanitising) procedure is carried out to "fix" the input. Special character escaping functions are a typical example of such sanitisations.

While such approaches are relatively easy to deploy, a problem may lie in the sanitising functions themselves. Proving that actual sanitising function implementations are correct can be non-trivial for complex cases (Doupé et al., 2013). If a bug is present in those functions, the protection would not work despite the efforts of continuous input monitoring. Another common issue of such run-time input sanitising is the (sometimes significant) performance cost introduced due to the sanitising overhead (Livshits and Chong, 2013).

Making the intrusion detection depend solely on the input parameters supplied by users might still be useful to detect malicious activity. However, such a view shifts the focus towards the users and does not provide any insight on the actual security of the application. For example, raising an alarm whenever an unexpected HTTP GET parameter is encountered allows detection of suspicious situations. Unless there is a bug in an application, the unexpected HTTP GET parameters can be treated as malicious activity, as the page generated by the application should only supply a predefined set of parameters back to the server. Therefore, if such an unexpected parameter is detected, it can be concluded that the parameter was supplied by the user and not through the intended means. While unexpected, such situation might not necessarily be an actual attack. Furthermore, even if the detected situation is an attack attempt, it may be impossible to predict whether the attack will succeed (or what actions should be taken to prevent the attacks, if any) without having a deeper understanding of how the application is organised. Some approaches perform initial analysis to derive a set of limitations to be imposed on user input to address this issue (Guha et al., 2009). Such enforcement might be used to ensure client authentication (Jovanovic et al., 2006a). Unfortunately, although such approach might increase security of the application, none of the security flaws are actually rectified in the application itself.

2.6. Application analysis

Some protection approaches focus on the application rather than on some behavioural aspects, in other words, the source of the application becomes the subject of analysis. However, due to the high complexity of thorough analysis for arbitrary source code, attention is usually paid to some specific aspects of the application only. For example, such application-oriented approaches commonly aim to detect code fragments which use user input in a sensitive way (such as supplying to a database as a part of SQL query) without properly sanitising it first. It is suggested that if all such incautious input data uses are detected and fixed (either manually or automatically) by applying appropriate sanitisation at the source code level, the application is then immune to user-input based attacks.

Application-analysing protection approach can also focus on detecting various discrepancies and inconsistencies to address logic-level application flaws (Felmetsger et al., 2010). In such an approach, the application is viewed as a set of input parameters and data flow paths (variable assignments). As with the growth of the number of data flow paths, it becomes more probable that developers may forget to include necessary sanity checks in some of the paths. A classic example is the number of items in the online shopping basket. If not all of the data flow paths check whether the number provided by user is positive, the user may be able to obtain unintended discounts.

Tracking all the data flow paths for a given user input allows detecting such logic flaws in an application given that at least one of the paths includes the necessary checks. The detected check can then be automatically enforced for the rest of the data flow paths to prevent the attacks based on unchecked user input. Unfortunately, while Felmetsger et al. (2010) claim that is an issue which can be solved with further engineering, it is not clear how the method should behave if none of the data flow paths contain the necessary sanity check. In addition, handling of different checks for the same data at the different flow paths is not covered. While in some cases combining two different checks might be possible, it is unclear of how the conflict should be resolved for mutually exclusive checks. Other, seemingly solvable limitations of the approach include the high memory usage and human assistance required.

2.7. Outputs analysis

Having no access to the sources of a web application, a subset of protection approaches may instead focus on the external behaviour of the application. Such output-oriented approaches mainly differ in the selection of application behaviour aspects they observe and analyse. The choice of behaviour aspects is limited by the ways a web application can interact with the external environment. Two of the most common ways of interaction used by web applications include HTTP protocol (request/response pairs) and database access. Less common interaction happens through file system access and generic network access.

While, observing the sequence of the HTTP responses alone can be useful for statistics-based decisions, it is common for protections to monitor application outputs in conjunction with inputs, perhaps even with addition of internal state or processing logic (Doupé et al., 2012; Li and Xue, 2011). Focusing only on one of the external aspects is typically effective against a limited subset of possible attacks. For example, observing just the HTTP responses may be adequate for detecting some of the attacks, but destructive SQL injection attacks cannot be detected in the same manner. Similarly, analysing only the database access actions can overlook XSS attacks. Focusing on even less frequent actions (file and network access) can allow more attacks to remain undetected. Thus, output-oriented protection approaches tend to monitor and analyse multiple, externally-observable actions at once.

2.8. Client-side considerations

A common view to protecting a web application is similar to building a fence around a protected object, and not trusting anyone around. However, the full picture of web interaction also includes legitimate web site visitors apart from the web application itself. The protection measures must therefore not only protect the web application itself, but also not interfere with client serving. Bringing visitors into the view allows categorising the protection techniques based on whether the protection subject is the web application server or instead is the users. For example, some XSS attacks do not target the web application, but are rather focused on other web site visitors. Therefore, an additional orthogonal classification axis can be based on the fact whether the client protection is considered in a given protection approach.

Server-oriented protection methods only focus on protecting the web application with the associated support infrastructure such as back-end database server or file system. Typical examples of server-oriented attacks include SQL injections, directory traversal flaws or logical bug exploitation. While some pure server-side approaches can secure web site visitors from such attacks as XSS (Nadji et al., 2009), that can be seen as a positive side-effect.

Client-considering approaches essentially solve the problem of protecting legitimate web site visitors from potential attackers. This category of protection considers the client security as well as the web application security. Two broad classes of client-targeted web-based attacks include social engineering and drive-by downloading (Chang et al., 2013). An attack succeeds when a client computer is forced to execute the malicious code. Social engineering based attacks trick the human into performing such execution manually, whereas drive-by downloads achieve the same goal in an automated way by the use of software vulnerabilities. While keeping the software updated might indeed protect against some drive-by download attacks, social engineering attacks are somewhat harder to deal with using purely technical approaches and often require user training as well.

Further sub-division of this category can be made to differentiate between protections implemented purely on server side and client-assisted approaches. Client-assisted approaches can ensure the consistency between the server-side generated document and client-side parsing and rendering (Nadji et al., 2009; Weinberger et al., 2011a), however, they are difficult to implement given an existing web browser infrastructure. Moreover, relying on client assistance is not an option of server-side protections, as malicious users can disable or alter the client-side code.

In summary, this kind of categorisation-protection approaches either secure the server side, or secure both servers and clients. The latter can be achieved both with and without client assistance. As purely client-oriented protection methods such as (Bates et al., 2010) are out of the scope of this study, they are not included in this classification.

2.9. Discussion

Figure 1 outlines the protection classification properties covered in this paper including the individual articles reviewed. As can be seen from the classification property distribution, the approaches involving intent-based decisions are rare. This can probably be attributed to difficulties related to intent extraction. Policy-based decisions are somewhat more popular than statistics-based ones, presumably because of ease of implementation and wide applicability. Approximately the same level of attention is payed to application and inputs combined with outputs.

Overall, as can be seen from Table 1, the general classification presented in this section is focused on how the protection method is implemented. While there are other ways commonly used to

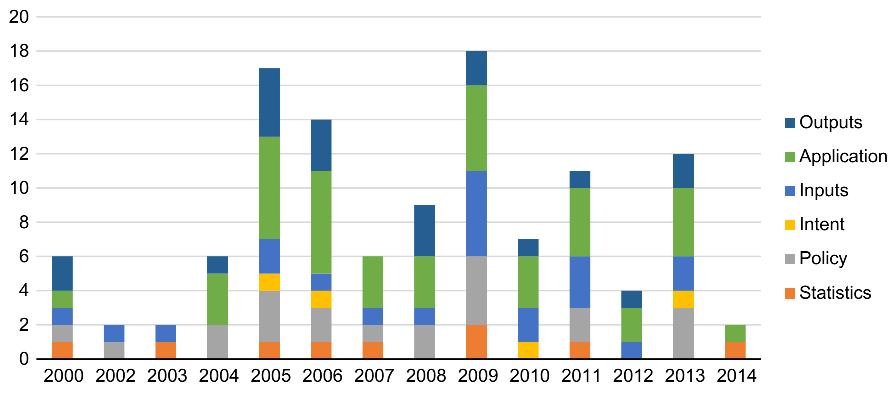

Fig. 1. Number of publications on protection classification properties from 2000 to 2014.

categorise existing protection technique, most are either not informative or not detailed enough. For example, describing a protection method to be either white-box or black-box does not actually clarify how the method is supposed to work. Such description can only be used as a quick filter to determine applicability of the method to a certain situation. Another typical example is dividing protections into taint-based and taint-free classes (Li and Xue, 2014). While technically correct, such division is somewhat limited, and not informative in describing taint-free protections. Other types of classifications may include such factors as placement of the protection system or time of detection. In addition, classifications based on strict hierarchies are not well suitable for describing hybrid approaches falling into several categories at once. Therefore, the proposed classification aims to clear up the confusions and inconsistencies between different types of classifications by providing a flexible and unified view. The next section discusses currently existing protection methods and their place in the proposed classification.

3. Existing web application protection techniques

A possible reason explaining the large amount of used protection classification methods may lie in different motivations driving protections development. Taking a step back, it can be seen that some protections emerge as an answer to a novel attack type (attack-specific protections), while others are designed out of a need to protect an existing application against hostile environment (application- or environment-specific). Thus, attack-specific protections are usually designed (or at least attempt) to be universally applicable to a variety of applications, whereas application-specific protections strive to be able to handle multiple different types of attacks. In addition to this difference, a protection developer may also have a pre-existing view on the way the protection should operate. Different protection developers may agree that some classes of attacks are only successful because of improper input validation; therefore a generic "input validation" protection category is suggested. However, despite protection authors having the same view, the actual protection implementations may significantly differ.

For example, based on the relation of the protection system to the source code of the application being protected, protection techniques can be broadly grouped into white-box and black-box approaches. A protection can either require access to the sources for preliminary analysis or implement the protection through external means only. White-box approaches involve source code analysis and are, thus, programming language-specific and mostly applicable for scripting languages. Conversely, black-box approaches do not usually depend on the programming language used, as the externally-observable application actions (e.g. HTTP request-response pairs, database access actions and system call invocations) are analysed instead (Scott and Sharp, 2002; Halfond and Orso, 2006; Mutz et al., 2006). A conceptual difference between the two approaches is that in the case of source code analysis, the intent of the application developer is considered. However, some aspects of the intent may be missed in the case of external observation.

Another common example includes grouping protection systems based on their placement. While a reverse proxy might protect against XSS attacks, some attacks such as destructive SQL injection attacks will remain undetected as the destructive actions are already taken by the time the generated pages reach the proxy. Analysing the cause of this shortcoming brings to a conclusion that for proxy-based approaches to work effectively, all of the output points of the application must be observed. The output points include all of the communications with the external resources, such as database, network or file system access attempts.

The same kind of limitation is applicable to log-based protection approaches. Some of the less dynamic protection techniques involve analysing application log files. The biggest advantage of both reverse proxy and log-based protection techniques is that they can be easily deployed in different environments. An important advantage of reverse proxy based approaches is that a proxy has more control over the data passed. For example, such a proxy may modify or completely drop the data passed to prevent an attack. For log monitoring approaches, however, it may be too late to prevent an attack by the time a malicious action is accessible in the log file. Therefore, log analysing is more commonly used for anomaly/intrusion detection rather than prevention.

The rest of the section reviews protections grouped based on some common trait used for categorisation by the existing classification approaches and is concluded by a summary table outlining the place of reviewed protection approaches in the proposed classification.

3.1. White-box approaches

Common trait: such protection approaches require access to the application source code

Describing a protection approach as a "white-box approach" is not very informative and can only be used to quickly outline the general applicability of the approach. Therefore, further details must be provided to distinguish between different white-box approaches.

For example, source code analysis employed by white box approaches can either happen prior to the code execution (static analysis) or at run time (dynamic analysis). Static analysis implies that the detected issues must be resolved before the application can be actually used in production environment. The results of such analysis can be regarded as recommendations on fixing the detected issues. While in some cases, the detected issues can be rectified automatically (Huang et al., 2004), using non-automated static code analysis can still be useful during the application development phase. Hybrid approaches may fix the detected issues automatically while still involving developers to be aware of the detected vulnerabilities (Medeiros et al., 2014). Dynamic analysis tracks the application code execution at run time and can respond in a more active manner (for example, by preventing execution of some code branches). Run-time analysis also allows the analyser to have access to user-supplied data. Such flexibility is commonly achieved by extending an existing programming language interpreter or standard libraries (Son et al., 2013; Cova et al., 2007; Halfond et al., 2006).

Another feature of run-time analysis is that having access to the source code allows building of an application's intended behaviour model. Such a model can be then used during the intrusion decision stage to check whether the exhibited application behaviour is as expected. White-box approaches may also focus on specific application aspects such as the APIs used (rather than analysing the application itself), atomicity constraints, approximated web page structure or exposed web API (Srivastava et al., 2011; Zheng and Zhang, 2012; Minamide, 2005; Halfond et al., 2009).

Regardless of the details of the analysis employed, due to the requirement of having access to the application sources, the use of white-box approaches is conceptually limited to either open-source software or applications developed in-house.

3.2. Black box approaches

Common trait: application source code access not required

Similarly to white-box approaches group, additional details are required to distinguish between various black-box approaches. A common distinction between black box approaches lies in the subject of observation. Viewing a web application as a black box means that protection approaches have access to only the inputs and the outputs of the application. Although hybrid approaches might observe both inputs and outputs, some might focus on one or the other only. Even more details are required to precisely distinguish between black box approaches observing only a subset of inputs or outputs.

The advantage of implementing black-box approaches is that such approaches can be deployed for any web application (even if the source code is unavailable). However, a deeper understanding of how the application should behave may not be possible with black-box approaches, potentially leading to a lower attack detection rate.

3.3. SQL injection protections

Common trait: such protections aim to circumvent or detect at least some types of SQL injection attacks

SQL injections are one of the most common attacks, thus a considerable number of protection techniques are developed specifically to circumvent SQL injection based attacks (Kemalis and Tzouramanis, 2008; Liu et al., 2009; Halfond and Orso, 2006; Bisht et al., 2010; Valeur et al., 2005). The actual approach details used in such protections vary based on the developer's view on the cause of the vulnerabilities; however, the SQL injection vulnerabilities are traditionally attributed to lack of proper input sanitisation.

Consider the following simplified PHP code excerpt:

$mails=mysqli_query($con, "SELECT MailSubject FROM Mails WHERE MailDate > '".$_GET["date"]."'");

Keeping the authentication issues aside, this kind of code could be used to retrieve a list of e-mails received after the specified date. The danger of this code lies in the seemingly harmless HTTP GET parameter 'date'. Unwanted consequences can arise if a web site visitor supplies the following value for date:

2000-01-01' OR 1=1

In that case, the following SQL query will be generated:

SELECT MailSubject FROM Mails WHERE MailDate > '2000-01-01' OR 1=1

This will cause the list to include all e-mail subjects. More destructive results can be achieved by using a different structure for 'date' value, such as including a semicolon character to issue more than one SQL command. While this is example of an insecure code, similar code fragments are frequently met due to historical reasons. Initially the security risks were not fully understood, and forming the SQL requests on the fly by concatenating strings looked simple and natural. In fact, some programming books even recommended that style of SQL request forming (Musciano and Kennedy, 2000). With the improvement of security practices, avoiding such vulnerabilities became possible with the use of escaping functions such as mysql_real_escape_string(). Applying such functions implicitly expresses the developer's intentions about the role of the input. Such function calls sanitise and make sure that the inputs will only be used as the predetermined type of data.

Input sanitising is a well-understood technique, which can prevent some of the attacks. This approach works best if applied properly during the development stage, as the developers have an opportunity to apply sanitisation before each sensitive database function call is performed. However, especially for complex applications, some of the database sensitive function calls might be missed due to human error, which can lead to vulnerabilities. Thus, various automated approaches were proposed. Request templating is a popular example of such approaches. The key idea behind templating is to completely eliminate unsafe string concatenation. To achieve that, the task of preparing the final SQL request is delegated to a specialised function. User inputs are passed to such function, which essentially sanitises them all before forming the final SQL request string. Prepared statements technique available in PHP is a popular example of such approach.

For legacy applications, an additional problem lies in identifying the sensitive function calls. Such identification is language-dependant and might involve complex static analysis (Zheng and Zhang, 2013).

Another problem common for both legacy and future software is to formally verify the correctness of the sanitising function (Yu et al., 2010). For multi-language web applications, even the order of sanitisation becomes important (Saxena et al., 2011).

SQLRand (Boyd and Keromytis, 2004) uses a non-traditional approach of changing the SQL request definition rather than preventing the malicious SQL request creation like other approaches. The SQL injection attacks succeed if the attacker can predict (at least partially) the form of the generated SQL request. Such knowledge allows the attacker to supply an input which alters the structure of the initially-expected SQL request. In general, injection attacks are essentially possible if an attacker has the knowledge of the result-language (HTML, SQL, etc.) syntax. The SQLRand approach, thus, aims to change the SQL syntax by randomising the language tokens. While such approach makes it harder for an attacker to guess the necessary input to form malicious SQL request, SQLRand essentially employs a security through obscurity approach. If the pseudo-random number generator used for randomising the SQL syntax is compromised, then the whole protection method fails.

3.4. Protections against XSS attacks

Common trait: such protections aim to circumvent or detect at least some types of XSS attacks

XSS attacks are made possible due to the fact that current web application development commonly involves mixing multiple languages together. This is reminiscent of the Von Neumann architecture, where data and code are stored in the same memory. For example HTML and JavaScript are commonly returned to a site visitor in the same stream. Insufficient context separation can lead to XSS attacks. XSS attacks succeed in cases when an attacker can supply data which is misinterpreted either by server side or by another web site visitor's browser. A typical example is the occurrence of $<$script$>$ … $<$/script$>$ construction in a text input area. If the supplied construction is inappropriately used to form subsequent web pages, the script supplied in the $<$script$>$ tag may be executed by other web site visitors.

XSS attacks are commonly categorised into two types: stored and reflected. Stored XSS attacks are persistent, as the maliciously injected scripts are permanently stored on the server side, for example in a database as a message posted to a forum. Whenever a web visitor retrieves a page containing the injected script, the visitor's browser might execute the malicious script. Reflected attacks are possible even if the injected script is not initially present on the server side. The attacks may succeed if the server responds with pages based on the user input. If the web page is not properly constructed, the attacker's input might be placed in the wrong context and subsequently executed on the client side. Due to the fact that for reflected attacks, the payload bounces back to the source, the attacker must trick users into sending the malicious script to the server first. Such tricking can for example be achieved by supplying the victim with a specially crafted link. An additional type of XSS attacks includes XSS DOM attacks. However, due to the fact that such attacks may not even reach the server side, protections against this specific type of attacks are out of the scope of web application protections.

While many XSS defence approaches are proposed, existing industry-standard solutions generally rely on user input sanitising (Wassermann and Su, 2008; Weinberger et al., 2011b).

As XSS attacks target clients XSS protections can technically be considered out of the scope of web application protection and are more a client responsibility. XSS protections, however, cannot be implemented purely on the client site as the browser has no prior knowledge about the intended structure of the web page. Therefore, web applications may choose to assist protections employed by the clients.

Both types of XSS attacks are made possible because browsers wrongly assume that web pages generated by server should be trusted, thus some researchers suggest that XSS attacks can be viewed as a trust problem, rather than context mismatch problem (Van Gundy and Chen, 2009; Nadji et al., 2009). Such trust-based approaches may for example involve signing the known trusted portions of the page, while skipping the user input. Looking deeper, it may be seen that trust-based approaches are using a white-listing model (specified portions are marked as trusted), while sanitising-based approaches are using a black-listing model (user inputs are marked as untrusted).

While most of the approaches focus on preventing the context escaping, some do not enforce any limitations on user input, an analysing reverse proxy is deployed to monitor the outgoing HTTP responses instead (Nadji et al., 2009). Such an approach allows the proxy to analyse the fully-generated web page, and, based on prior knowledge of the intended web page structure, either let the response reach the web site visitor, or drop the response completely if an attack is detected. Such approaches might seem simpler to implement, because of the lack of necessity to sanitise the user input. However, the complexity is essentially shifted into obtaining an accurate description of the intended web page structure.

3.5. Embedded SVG attack and protection

Common trait: such protections aim to circumvent or detect embedded SVG attack

The increasing complexity of modern web technologies potentially will introduce more risks. The ability to include complex and interactive objects, such as SVG images or Flash content, leads to more attack paths. Considering that SVG images might contain JavaScript code, it can be dangerous to accept and embed SVGs from unknown sources (Heiderich et al., 2011). One of the possible protections would be to validate the SVG against an XML schema. However, as shown by the authors, such validation is not enough, as XML schemas focus on the structure of the XML document, rather than on the content.

Another approach is SVG purification, which involves removing all potentially malicious content from the image file, while minimising the visual impact. Purification is achieved by whitelisting the allowed content. While the approach seems to work adequately enough, it can also be addressed at HTML specification or browser level. For example, the current HTML5 spec explicitly states that interactive SVG content should not be executed by the browser (World Wide Web Consortium (W3C), 2014).

A possible solution would be limiting web browsers from automatically executing interactive content. However, the diversity of browsers available on the market (and perhaps compatibility reasons) makes it difficult to implement such restrictions in practice. Even if widely adopted, such a solution shifts the problem to the domain of social engineering; as such attack could still succeed if the users are tricked into allowing the blocked interactive content execution manually.

3.6. Input monitoring protections

Common trait: such protections assume that observing user inputs is adequate to detect some types of attacks

Such protection systems consider the application to be a black box and focus on the input data (Buehrer et al., 2005; Wassermann and Su 2004; Haldar et al., 2005). The three types of commonly used inputs in HTTP protocol are HTTP GET parameters, HTTP POST parameters and HTTP headers. The intuition behind such approaches is that some characteristics are different for inputs causing attacks and inputs supplied during the normal every-day usage. Thus, user inputs are constantly monitored and analysed, in case a non-typical input is detected, whereupon an alarm is raised. Some approaches may additionally observe other factors such as web page DOM structure (Bezemer et al., 2009). Unlike reverse proxy-based approaches, user input monitoring allows detection of potential attacks at an earlier stage, which makes it possible to both detect and prevent attacks.

The key problem for user input monitoring approaches lies in selecting the correct characteristics to observe. Multiple characteristics are often used to improve the detection rate and reduce the false positive rate. Generally, there is no obvious strict difference between normal and malicious user inputs; therefore, the choice of characteristics to observe is guesswork to some extent. A protection system may be designed to monitor a multitude of factors in hope that a non-obvious correlation with maliciousness of the input is automatically discovered (Kruegel and Vigna, 2003). The characteristics to observe are not chosen randomly but are rather based on implicit assumptions on the significance of the characteristic deduced from previous experience. If a characteristic is known to differ for some attacks, it is reasonable to include that characteristic to the list of factors to consider. However, such assumptions may not be true or adequate for real-world deployment.

Some examples of such significant input characteristics include the input length, input character distribution, absence of the input and so on. For example, analysing the length of the input may be justified by the fact that buffer overflow attacks require longer inputs to exceed the length of the allocated buffer. The necessity of analysing the input character distribution can be explained by the fact that some types of attacks (such as SQL injections) may involve using rare symbols, not commonly encountered during everyday usage.

To simplify the deployment on different systems, the training phase is used on per-application basis, to discover what the normal user input is. The protection system essentially learns what kind of information users normally supply during every day usage. This captured information is subsequently used during real-time monitoring phase to determine whether currently supplied user input differs from the "usual" inputs. However, determining the difference threshold to raise an alarm is a non-trivial problem of its own, not having a precise answer which commonly leads to probabilistic detection results. Overall, such approaches suffer from the same problems inherent to training-based approaches – providing a clean training environment, covering all usage scenarios and managing non-precise detection results.

3.7. HTTP parameter pollution

Common trait: such protections aim to circumvent or detect at least some types of HTTP parameter pollution attacks

A narrow type of attack relies on a higher level server-side implementation-specific flaws of application environment – namely the language interpreter, third-party APIs or the HTTP server related flaws rather than the application-level bugs. The HTTP parameter pollution attack is possible because the HTTP protocol specification only defines the structure of the input parameters (supplied by users) without imposing enough restrictions on the actual parameter names and values. At a lower level, the HTTP request received by the server is merely a string of bytes, which has to be parsed by the software to form a convenient data structure usable by the web application. Such preliminary preparation commonly includes creating a dictionary structure containing the parameters passed. A problem arises if an HTTP request contains two or more parameters with the same name. While such occurrence does not contradict the HTTP protocol definition, the dictionary data structure cannot contain the same key more than once (Balduzzi et al., 2011). Thus, depending on the implementation, some of the parameters may be completely lost and never reach the web application itself.

Malicious users may form HTTP requests having multiple GET parameters with the same name, leading to ambiguities at the server side. Some server-side software picks up the first occurrence, while others pick up the last. In specific cases all the values supplied might be picked up, propagating the ambiguity to the web application itself. The HTTP parameter pollution attack becomes possible if the sanitising function is only applied to one of the passed parameters, while it is the other occurrence which is actually used to generate the response.

The problem exploited by HTTP parameter pollution attacks is essentially a wrong implicit assumption about the way HTTP GET or POST parameters are formed into an HTTP request and passed to the server. For example, even if a legitimate web browser is designed to never emit an HTTP request containing multiple parameters with the same name, nothing prevents an attacker from crafting such malicious requests manually. This means that a web server cannot impose or assume any restrictions on the received HTTP request.

While it is possible to design (or modify existing) web applications to be resistant to HTTP pollution attacks, such modifications have to be highly dependent on the application environment (e.g. example web server version or the scripting language used). The variety of web development frameworks available on the market may complicate such modifications even further. Overall, the HTTP pollution attacks largely depend on bugs and inconsistencies present at a different level then the web application itself. Therefore, a universal solution to the problem would involve clarifying the web standards (such as HTTP GET/POST parameter handling procedures). This, however, would still require patching the existing web development tools and execution environments.

3.8. Improper user input handling

Common trait: such protections assume that enforcing a uniform and correct user input handling is adequate to detect or circumvent some types of attacks

Data sanitising is a commonly enforced user input handling technique. In the early stages of web development, user input sanitisation was only possible on the server side. With the advancement of web browser capabilities, it became possible to perform more tasks on client-side. User input verification is commonly implemented on the client side to improve the user experience, as the client-side user input checking is significantly faster than server-side checking. Such a difference is explained by a noticeable packet round trip time to and back from the server. Thus, an inadvertent mistake in filling a form on a web page degrades user experience by introducing unnecessary delay. In short, the client-side input verification is only used for user convenience, while for security purposes only server-side verification can be used.

In the simplest form, a problem arises if only the client-side user input verification is implemented. Overlooking the server-side implementation allows an attacker to simply disable JavaScript or manually forge an HTTP request to bypass the security checks. But even if the server-side checks are deployed, the actual implementations are commonly different, as different languages (such as JavaScript and PHP) are generally used on the client and server side. Such discrepancies may lead to attacks. Thus approaches involving comparison of client- and server-side sanitisation procedures have been proposed (Bisht et al., 2011; Skrupsky et al., 2013; Srivastava et al., 2011). These proposed protections aim to find and report discrepancies between sanitisation procedures performed on client and server sides. If a server-side sanity check is less restrictive than the corresponding client-side check, an attack might be possible as malicious users of web applications can disable the more restrictive client-side sanity check (e.g. by turning off JavaScript support).

Taint tracking is a more specific technique commonly used to analyse user input handling aspects of the application (Nguyen-tuong et al., 2005; Chess and West, 2008; Chin and Wagner, 2009; Livshits et al., 2009; Tripp et al., 2009; Jovanovic et al., 2006c). As the set of sensitive functions (database access functions, system calls, etc.), is known beforehand and user inputs can only be passed through a predefined set of means (GET or POST parameters, cookies or headers), the main task of taint-based approaches is to track data flows to detect situations when user input is passed to sensitive functions. Taint tracking is accomplished by variables assignment analysis. Depending on an actual implementation, a different depth of analysis may be used – some studies consider taint at variable level, while others may deepen the analysis by considering individual string variable characters (Halfond et al., 2006). A precise taint tracking might not be achievable for highly dynamic cases if only the application source code is being analysed. For this reason, approximation-based techniques were proposed (Yu et al., 2010). For example, such approximation techniques can generate context-free grammars or finite state automata, which can be used at run-time to detect whether the SQL queries issued by the protected application are valid and allowed.

In some cases, the actual sanitising process may even be out of scope for application analysing approaches, as the task of sanitising can be delegated to the sanitising function previously implemented in the source code (Saxena et al., 2011). In such scenarios, the protection is just ensuring that sanitisation procedure is carried for all cases of user input usage and no sanitisation is implemented by the protection scheme itself.

Traditionally, the tainting approaches focus on untrusted data (negative tainting) (Nguyen-tuong et al. 2005; Livshits and Chong, 2013; Tripp et al., 2009), but an alternative approach of tracking only the trusted data (positive tainting) can also be used (Halfond et al., 2006). The advantage of positive tainting is that an unidentified trusted variable can result in a false positive detection, which is not a critical problem, as it can be quickly corrected. In case of negative tainting, a missed untrusted variable can lead to a false negative, which is undesirable.

A comprehensive sanitisers analysis revealed two novel types of errors (Saxena et al., 2011). The first kind of error occurs when multiple sanitising functions are applied in a wrong order. Depending on the contexts and sanitisers involved, the actual parsing performed by a web browser might lead to malicious code execution. The second type of error occurs when the sanitiser function is applied in the wrong context.

To address such kinds of errors, the following two-step procedure is used. First, untrusted data is identified by dynamically analysing the execution of the application being protected. The analysis consists of tracking the trusted data flow, i.e. positive tainting approach is used. Due to the high overhead of the analysis, the results of the analysis are stored in a cache for future use. Second, the auto-correction module is patching the flawed execution paths to apply proper sanitisation procedures.

Although the process does not require application code changes, an expensive analysis process must be taken if the application source changes, which may introduce performance issues in highly dynamic environments.

3.9. *User input isolation*

Common trait: such protections assume that separating user input from other data processed by the application is adequate to detect or circumvent some types of attacks

While focusing on user input is beneficial in terms of early detection, the lack of context separation between different kinds of data (such as user input and application-generated content) and code complicates user input analysis. The result generated by a web application is merely a string of bytes containing no semantic or structural information. In fact, the generated result may be not an HTML page altogether but, for example, a compressed zip file. Although, in case the result is a web page, syntax parsing may allow one to separate various structural parts of the string such as JavaScript, HTML or CSS blocks. Unfortunately, no structural information on the source of the sub-strings is stored anywhere in the generated page itself.

As the trust to the returned data is first based on the source of the data, adding meta-information to separate user input from the web application output can allow one to distinguish between the two while analysing the generated string (such as web page or SQL query). Given no additional data stream, such meta-information must be stored as the part of the string itself. Using a set of protective symbols to surround the user-supplied string portions is a natural data separation approach (Su and Wassermann, 2006; Bravenboer et al., 2007).

The problem of such approach is that the protective symbols are part of the same string character domain. If the symbols used for separation are known to the attacker, the attacker is free to supply the same set of symbols to gain control over context switching. In some cases, even brute-forcing could be enough to gain the knowledge about the symbols used (Su and Wassermann, 2006).

3.10. *Access control based protections*

Common trait: such protections assume that formal access control is adequate to detect or circumvent some types of attacks

Another traditional class of protection approaches is based on notion of roles and corresponding role access enforcement. This kind of protection is highly application-specific, as different applications have different sets of roles. Therefore, a fully automated generic access control solution might not be achievable. The initial definition of roles along with the corresponding external resources they access must be provided by the developer either during the development stage or at the time the protection system is deployed. That information is essentially part of the developer's intent, and for example, might include sets of web pages accessible by each role. After the initial definitions are in place, such approaches can use various techniques to automatically derive access attempts by, for example, scanning web pages for various features such as hyperlinks or any specific object access attempts (Sun et al., 2011; Schneider, 2000). Overall, this class of security measures is focused on a specific aspect of application behaviour and cannot detect misuses of the explicitly-accessible resources.

3.11. *Summary*

Table 2 summarises the protection techniques discussed in this section.

As can be seen from Table 2, intent-based decisions are somewhat underexplored. Applications gain the same level of attention as inputs and outputs combined. Policy-based decisions are dominating, presumably because ease of implementation. A significant amount of papers reviewed lack the decision component, which is typical for vulnerability scanners. Interestingly, none of the reviewed protections methods attempts to combine different decision bases.

4. Current issues and limitations

This section focuses on various issues of the currently existing protection techniques covered in the previous Section. Some of the problems are inherent to multiple approaches, for example the manual work required to deploy the system, while others are approach-specific such as lack of support of object-oriented features of PHP (Jovanovic et al., 2006b; Sun et al., 2011). The inability to actually prevent attacks might be regarded as a limitation for attack detection systems compared to attack prevention systems. However, this section discusses limitations which are common to both attack detection and prevention classes of protection.

4.1. *Manual work required*

A significant disadvantage for some of the approaches is the requirement to perform parts of security protection deployment manually (Jovanovic et al., 2006b; Pietraszek and Berghe, 2005; Kemalis and Tzouramanis, 2008; Louw and Venkatakrishnan, 2009). While such a condition can be fulfilled for development-practices oriented approaches, the additional time required for

Table 2
Classification of protection techniques reviewed.

Paper	A. Statistics	B. Policy	C. Intent	1. Inputs	2. Application	3. Outputs	Class
(Doupé et al., 2012)				✓		✓	{1 3}
(Doupé et al., 2013)		✓			✓		{B 2}
(Nguyen-tuong et al., 2005)		✓			✓		{B 2}
(Liu et al., 2009)		✓		✓		✓	{B 1 3}
(Guha et al., 2009)		✓		✓			{B 1}
(Livshits and Chong, 2013)		✓		✓	✓		{B 1 2}
(Livshits et al., 2009)					✓		{2}
(Chess and West, 2008)				✓	✓		{1 2}
(Kruegel and Vigna, 2003)	✓			✓			{A 1}
(Bezemer et al., 2009)		✓		✓	✓		{B 1 2}
(Mutz et al., 2006)	✓					✓	{A 3}
(Scott and Sharp, 2002)		✓		✓			{B 1}
(Balzarotti et al., 2008)		✓			✓		{B 2}
(Balzarotti et al., 2007)					✓		{2}
(Ray and Ligatti, 2012)					✓		{2}
(Chin and Wagner, 2009)		✓		✓	✓		{B 1 2}
(Merlo et al., 2007)		✓		✓	✓		{B 1 2}
(Yu et al., 2010)				✓	✓		{1 2}
(Sun et al., 2011)					✓		{2}
(Maggi et al., 2009)	✓			✓	✓		{A 1 2}
(Schneider, 2000)		✓			✓	✓	{B 2 3}
(Valeur et al., 2005)	✓					✓	{A 3}
(Wassermann and Su, 2004)					✓		{2}
(Wassermann and Su, 2008)					✓	✓	{2 3}
(Buehrer et al., 2005)			✓		✓	✓	{C 2 3}
(Medeiros et al., 2014)	✓				✓		{A 2}
(Skaruz, 2009)	✓					✓	{A 3}
(Kemalis and Tzouramanis, 2008)		✓				✓	{B 3}
(Balduzzi et al., 2011)				✓			{1}
(Cova et al., 2007)	✓				✓		{A 2}
(Grechanik et al., 2006)					✓		{2}
(Samuel et al., 2011)		✓			✓		{B 2}
(Skrupsky et al., 2013)		✓		✓		✓	{B 1 3}
(Jovanovic et al., 2006a)		✓		✓		✓	{B 1 3}
(Jovanovic et al., 2006b)					✓		{2}
(Jovanovic et al., 2006c)					✓		{2}
(Tripp et al., 2009)					✓		{2}
(Saxena et al., 2011)		✓			✓		{B 2}
(Bisht et al., 2010)			✓		✓	✓	{C 2 3}
(Bisht et al., 2011)				✓	✓		{1 2}
(Artzi et al., 2008)						✓	{3}
(Son et al., 2013)			✓		✓	✓	{C 2 3}
(Boyd and Keromytis, 2004)		✓			✓	✓	{B 2 3}
(Pietraszek and Berghe, 2005)		✓		✓	✓	✓	{B 1 2 3}
(Livshits and Lam, 2005)					✓		{2}
(Felmetsger et al., 2010)				✓	✓		{2 3}
(Haldar et al., 2005)		✓		✓	✓		{B 1 2}
(Lee and Stolfo, 2000)	✓			✓		✓	{A 1 3}
(Halfond and Orso, 2006)			✓		✓	✓	{C 2 3}
(Halfond et al., 2006)		✓			✓		{B 2}
(Li and Xue, 2011)	✓			✓		✓	{A 1 3}
(Huang et al., 2004)		✓			✓		{B 2}
(Minamide, 2005)					✓	✓	{2 3}
(Xie and Aiken, 2006)					✓		{2}
(Zheng and Zhang, 2013)					✓		{2}
(Zheng and Zhang, 2012)					✓		{2}
Total occurences	**9**	**21**	**4**	**21**	**41**	**20**	**19 classes**

manual work may make it infeasible to deploy the protection in some environments. Manual work is commonly required for developer-assisted approaches; however, it may also be implicitly present for some seemingly automated approaches as well. For example, while rule-based approaches are mostly perceived as automatic, the task of updating the rule set might require considerable ongoing effort. Such effort might not seem time-consuming given the modern approaches of automatic updating available in the existing software allowing effective work reuse. The rule set is prepared manually by the software manufacturer once, and is then propagated to the client software through the internal update feature. Nevertheless, the manual part of the work must still be performed for both update servers and client-side custom fine-tuning of the protection system.

In some cases, the manual assistance is not strictly required for the protection to function; however, such assistance can be used as an option to improve the detection rate. The optional developer assistance is commonly referred to as *hinting*. In case of ambiguities, such hinting is intended to improve the protection system's understanding of the web application being protected. Hinting is usually applied by modifying (marking) fragments of the source code in a pre-defined way. The purpose of marking the code fragment is either to attract attention of the protection system to the marked code fragment or to make the protection ignore the code fragment (distracting hinting). The choice is tightly tied to whether the protection system produces more false negatives or false positives. In a situation when a protection misses some of the dangerous events (false negative), identifying and marking the corresponding code fragments to be treated in some specific manner is a natural way to improve the detection rate. In the opposite case, when a protection system incorrectly identifies normal (harmless) events as attacks (false positive), the corresponding code is specifically marked to be treated as non-harmful under some conditions. Distracting hinting might be more secure, as false positives provide less risk, albeit at the cost of convenience.

Overall, such a requirement for manual work is quite generic and is applicable to various approaches regardless of the protection method employed. Apart from increasing the cost of the deployment, manual effort may also introduce errors due to the human factor, possibly degrading the attack detection rate. Consequently, an effective protection systems should strive to minimise or, even better, eliminate the need for any manual work. Nevertheless, allowing manual intervention as a possibility may improve the attack detection rate for complex applications where automated approaches are unable to fully and correctly deduce the application author's intentions.

4.2. *Unknown attack detection latency*

A limited ability to detect previously unknown attacks is another important limitation inherent to all rule-based detection approaches. Such protection systems fail with the emergence of new attacks which are not detectable by the deployed rule set. Unknown attacks are commonly referred to as zero-day attacks. The protection system users are at risk during the period when the rule set is not yet updated to cover the new attacks and deployed. The same kind of problem is similarly applicable to traditional signature-based antivirus protection systems. An advantage of such strict rule-based detection is a relatively low level of false positives.

The zero-day period (latency) is inherent to all reactive protections and many attempts focus on reducing the latency. However, a more effective approach is to implement proactive protection. In contrast to reactive protection, proactive protection techniques are designed to detect (and possibly prevent) previously unknown attacks as well.

4.3. *Detection factors selection*

As relying on a rule set is not applicable for implementing a proactive protection, other factors must be taken into consideration to allow attack detection. Such factors are typically captured by the monitoring module and fed into the analyser module to perform the actual detection. Regardless of the technical details of the final decision phase, the analysis must rely on significant factors as making decisions based on insignificant or irrelevant factors will not improve the attack detection rate. Thus, selecting a relevant and significant set of factors to monitor is vital for efficient attack detection.

However, it is difficult to formalise such a selection. The choice of factors made by various protections is based more on the developer's previous experience rather than on strict formal criteria and reflects the different challenges faced by different sites. A common detection improvement technique is to select a multitude of factors hoping that quantity will eventually grow into quality. For example, multiple HTTP parameter factors can be selected simultaneously, such as length, a fixed prefix and even character distribution (Kruegel and Vigna, 2003). While the selection is not completely random and each of the factors has a reason to be included in the set (usually based on some specific attack), no formal selection process is introduced. Moreover, in some cases being aware of the factors selected for a specific protection system can assist an attacker to bypass the protection. For example, relying on character distribution to distinguish English text and shell code is not enough as there are known ways to encode arbitrary code to have the same character distribution as English text (Mason et al., 2009). In some cases, using string length as an additional factor may help, as such English text mimicking encoding process increases the string length. However, in environments where an average string length is large enough, such additional factor might not be enough.

Ignoring some aspects of an application behaviour may lead to vulnerabilities. For example, approaches using reverse-proxies to intercept and analyse a generated web page cannot detect destructive SQL injection attacks, as by the time the interception happens all the SQL queries are already issued by the application. Thus, while such reverse-proxy based protections may detect leaking information, any destructive SQL actions will remain unnoticed. Although still useful as client-side protection, this limitation makes the use of reverse-proxies impractical for server-side protection.

In addition, a question of factor overlapping arises. While it might be tempting to select as many factors as possible, some of them might actually be covered by others. Such overlapping means that including more factors may not improve the detection rate and instead might actually degrade the performance of the detection because more factors needed to be processed.

The selection process is more complicated for black-box approaches, as the choice of factors is made based purely on the protection system developer's expectations. However, the actual web application developer's intentions are not taken into account. For white-box approaches, the source code itself becomes an additional factor to be analysed by the protection system. Static white-box approaches use the source code as the only factor, while dynamic white-box approaches use the source code factor in addition to other factors (possibly observable by black-box approaches as well). Such additional factors reflect the developer's intended behaviour of the application. Therefore, bringing the source code into the picture may improve the attack detection rate. However, despite having seemingly same goals on the surface, black-box and dynamic white-box approaches are solving fundamentally different problems. As an ideal result, black-box approaches can only guarantee (by detecting deviations) that an application is working in accordance to a model created by the protection author. Conversely, while varying in the level of detail of the deduced model, dynamic white-box approaches can guarantee whether the actual application behaviour matches the developer's intents captured by the model. Static white-box approaches using the source code as the sole factor to analyse are essentially evaluating the deduced application behaviour model against best development practices. The discrepancies between application developer and protection developer models may render the protection ineffective.

4.4. *Sanitisation verification*

User input sanitising aims to limit the possible application inputs to conform to a predefined set of rules. While such limiting is easily achievable for simple rules such as “only allow numbers” or “only allow valid post codes”, the sanitising becomes complicated for free form text inputs. With the increased complexity of the structure of the allowed input strings, the sanitisation process itself becomes accordingly complicated and requires a formal verification to be trusted (Balzarotti et al., 2008).

Additional problems arise due to multiple data interpretation contexts inherent to web development. Such a variety of contexts may require implementing multiple sanitisation functions which can lead to sanitisation mismatch vulnerability (Saxena et al., 2011). More specifically, if a wrong sanitisation function is applied to a given user input, unwanted string characters may not be filtered out and may remain in the context where the input is actually used. Furthermore, applying more than one sanitisation function (for different contexts) to the same user input might still lead to a vulnerability, as it was shown that in some cases such sanitisation functions are not commutative (meaning sanitisation for different contexts must be performed in a specific order only) (Saxena et al., 2011).

4.5. *Technical issues*

Some of the limitations applicable to various protection techniques are of a technical nature and can be removed by more engineering effort. A typical example would be the lack of support of PHP objects and classes for some approaches (Sun et al., 2011). More generally, supporting just a single programming language for the static analysis is another limitation by itself. However, implementing language-independent protections obviously requires more efforts. Thus language-specific protections commonly target popular programming languages such as PHP, Java or ASP.NET (Cova et al., 2007; Artzi et al., 2008; Saxena et al., 2011; Xie and Aiken, 2006; Livshits and Lam, 2005).

The same kind of limitation is applicable for the client-side code. Due to high demand for dynamic (responsive) web content, some functionalities may be offloaded to the client side with the use of JavaScript and Ajax technologies. Bringing a new language into the view creates more possible contexts for the source code generated and complicates the code analysis (Guha et al., 2009). Thus, approaches ignoring the added context may miss some of the aspects of the behaviour of the web application, possibly reducing the detection rate.

Source code analysis based protections also encounter the problem of the analysis depth. Some simpler approaches may, for example, use regular expressions to detect hyperlinks (Antoniol et al., 2004). The advantage of simple analysis is the speed, however given multiple nested contexts such straight-forward searching might be inadequate in some cases (Balzarotti et al., 2007). Overall, deeper context-aware analysis can potentially provide better understanding of the way the web application is supposed to behave.

4.6. *Detection precision*

Due to the nature of some of the protection techniques the detection is inherently imprecise (Kruegel and Vigna, 2003; Mutz et al., 2006). Thus alarm-triggering thresholds are introduced. While defining thresholds may be effective enough for most cases, a probability of false negatives could be increased. A notable cause for such detection uncertainties lies in the difficulties of covering all of the possible normal every-day usage scenarios. Anomaly-based protections compare the currently observed behaviour to the behaviour exhibited by the application during the training phase (normal behaviour). However, the normal behaviour is commonly defined statistically, rather than formally (Mutz et al., 2006).

Furthermore, detection drifting might be possible for self-learning protection techniques. In dynamic environments the application usage changes over time, thus the protection is required to constantly adjust the normal usage profile. An attacker might be able to perform the attack by slowly retraining the protection system to consider the attack to be a normal usage of the application (Bai and Kobayashi, 2003). Considering that in some cases, the normal usage of the application might change overnight, the detection decision may be complicated. Such drastic usage change is not commonly considered to happen for typical desktop applications. However, for web applications such changes might happen every time the web application is updated. If some functionality is introduced or removed from a web application, the usage behaviour observed by the protection system may change significantly even if the changes are not visible to the web application users. Thus, such significant changes to the web application may invalidate the protection system's previous knowledge about the normal usage, which in turn requires the developer to re-run the training phase. In summary, training-based protection methods may not be very effective in highly dynamic environments, especially if the training phase is lengthy and the application being protected changes significantly.

4.7. *Performance considerations*

Protection performance may become an issue in highly loaded environments as more data has to be analysed. The most obvious performance characteristic for a web application is the additional latency introduced by the protection system. Web applications themselves have considerable latencies explained by the nature of client-server architecture. Although web development techniques aim to minimise such latencies in general (commonly using AJAX technology), the actual underlying HTTP connection latencies may still be significant depending on the quality of physical connections involved (including the client to server distance). Thus, web application protection approaches are commonly optimised to introduce latencies which are insignificant in comparison with the actual HTTP latencies (Huang et al., 2004; Buehrer et al., 2005; Livshits and Chong, 2013).

Another characteristic indirectly impacting web application performance is the memory usage associated with the protection system. Some types of analysis might require storing considerable amounts of intermediate data structures (Felmetsger et al., 2010). Such limitation may make a protection system impractical. Apart from straightforward methods of improving the actual algorithm used, hardware-assisted approaches can be used. For example, while software-based string matching may not be achievable for high-speed networks, hardware-assisted signature detection based on Bloom filters can be used (Dharmapurikar et al., 2004).

4.8. *Other limitations*

One of the less common issues includes the use of security through obscurity in some of the approaches (Boyd and Keromytis, 2004). For example, in SQLRand, an additional context layer based on a randomised set of SQL instructions is introduced to complicate context mismatch based attacks. Thus, an attacker is unable to inject SQL code to meaningfully tamper SQL queries structure. However, the approach fails to work if the random number generator used is compromised. Another danger lies in the possibility of brute-force attacks. If an attacker is able to deduce the random to normal SQL instructions mapping, the protection fails as well.

Another problem of some protection techniques is the necessity of browser collaboration (Louw and Venkatakrishnan, 2009). Involving the client side in the protection may be helpful against client-oriented attacks (e.g. XSS). However it is risky to rely solely on client assistance for server-side protections as attackers have more freedom in investigating and modifying (or completely disabling) the client-side protection parts. A side-effect of relying on client-side code is that a generic solution must be implemented for all popular web browsers, which can be quite time consuming given the variety of web browsers available on the market. Such an implementation may also take a considerable time to be widely adopted. Deeper analysis of such approach uncovers another problem substitution: instead of generating a clearly structured web page, an additional stream of instructions on how to process the page must be provided.

Lastly, a problem of analysing intentionally unsafe web applications is somewhat under-explored, some approaches completely ignore such aspects (Halfond and Orso, 2006; Kemalis and Tzouramanis, 2008). While not commonly exposed to regular web site visitors, such potentially unsafe web applications are frequently used by web site administrators. A popular example is PHPMyAdmin database administering tool. The problem is complicated even further considering the capacity of the tool to send arbitrary user-supplied SQL queries (possibly destructive). Although such a capacity is potentially unsafe, it is an intentional functionality implemented by the application developer. Thus, a hinting mechanism to allow developers to explicitly mark portions of code as intentionally unsafe might be useful.

5. Discussion and conclusion

Observing the general picture, some similarities between seemingly different attack types can be noted. For example, XSS attacks are in their core very similar to SQL injection attacks. Both are possible because of context mismatches. In case of SQL injections, user-supplied data is wrongly put in SQL command context by for example escaping the string context or command boundary; for XSS attacks, the user input escapes HTML tags intended by the author. The root of the problem lies in the lack of separation of data and (sometimes multiple) code streams. The misinterpretation of data context can also be explained by the perception difference between the developer and a web browser.

An example of this difference for a web page fragment (central column) is illustrated in Fig. 2. A browser interprets the fragment as having four context transitions. However, the web page developer intended the fragment to have only two context transitions. The first intended transition is the change of context from HTML markup to static text, and the second being the reverse context switch back to HTML markup after the static text ends. However, if a malicious web page visitor manages to inject <img src="..."/> in place of a legitimate user name, two unexpected context transitions are introduced. The consequences vary depending on the actual type of context transitions. For example, an unexpected text to HTML transition may lead to an XSS attack.

Despite the similarities between the roots of XSS and SQL injections attacks, it may be hard to apply the same protection against both due to technical difficulties. For example, detecting XSS attacks can be performed by analysing the HTTP responses, but for SQL injections, other sources such as PHP code analysis or run-time database request monitoring are required. Given the root cause similarities for these kinds of attacks, a unified approach would be beneficial.

Such context mismatch based attacks are possible because of lack of user-supplied data isolation. Some approaches achieve such data isolation by introducing an additional abstraction layer, which treats user-supplied input in a special way (Doupé et al., 2013). Unfortunately such approaches are difficult to apply to existing legacy web applications. Another limiting factor is the necessity to learn new programming techniques introduced by such approaches.

Taking a wider view it can be seen that user-supplied data isolation is not only inherent to web applications, but for desktop or other types of network applications as well. For example, stack overflow attacks succeed when user-supplied data is misinterpreted as code and subsequently executed. While the ability to separate data and code is a feature inherent to the Harvard CPU architecture, such a separation can be introduced for Von Neumann CPU architecture for security reasons. A common protection against stack overflow attacks involves the use of CPU-specific technology to mark specific memory regions as non-executable (Marco-Gisbert and Ripoll, 2014). Such additional security measures can be effective in some cases, however the use of interpreted programming languages commonly met in web development makes such protection inapplicable. The complications arise because of the additional abstraction layers introduced by the interpreter itself, the crucial difference being a different memory view. At a lower level the data execution is prevented by setting a non-executable (NX) bit for a whole memory page. However, as programming language interpreters tend to provide portability and hence aim to be CPU-independent, such low-level notion as memory pages may be not exposed to the programmer at all. Simply said – the CPU level NX bit would not help securing PHP web applications because both PHP code and HTML data are treated as data from the point of view of the CPU. In contrast, PHP code is treated as code and HTML data as data by the PHP interpreter.

Web browser interpretation	Web page fragment	Developer intention
HTML: markup	`<b>`	HTML: markup
Text	`Welcome`	Text: greeting message
HTML: image	`<img src="..."/>`	Text: user name
Text	`!`	Text: greeting message
HTML: markup	`</b>`	HTML: markup

Fig. 2. Web page perception difference.

In the context of interpreters, a possible solution would be to introduce data and code separation at the interpreter level, for example, treating all string variables as objects having a source attribute and applying tainting techniques to keep track of the source of a given string. Such source information will be lost if the string is simply transferred over a network. Thus, rather than introducing a new data transfer method (which might be complicated due to the compatibility reasons), the generated web page should be analysed on the server side first. In case the analysis does not reveal any context violations, the generated page is passed further to the client side. Otherwise, the generated page should be dropped and an alarm could be raised.

An important advantage of such an approach is the ability to automatically retrieve the intended web page structure. The developer intentions are a critical part of the design of an application. Programming bugs commonly found in applications are essentially a mismatch between the developer's intentions and actual application behaviour (unless the developer makes mistakes on purpose). However the developer's intentions may be completely overlooked by black-box protection approaches. Such neglect is typical for anomaly detection based protections which aim to determine whether an application behaves "as usual" rather than "as intended". An alarm should still be raised if an application behaves as usual, but not as intended. Unfortunately, detecting intentionally malicious functionality is a more complicated process requiring a formal definition of what malicious functionality is. Even human brains might sometimes fail to recognise dangerous code for long periods of time (Imran et al., 2014) making it unlikely for reliable, fully-automated solutions to emerge in the nearest future.

Another advantage in contrast to probabilistic detection approaches would be precise detection. Observing a behaviour of an application and having access to the source code it is relatively easy to confirm whether the behaviour observed was intended. Thus, unlike for probabilistic approaches, a precise decision is possible if the developer intent (derived from the available source code) is taken into account.

Table 3
Comparison of protection techniques highlighting advantages and disadvantages.

Classification property	Advantages	Disadvantages
Statistics	May be able to detect unknown types of attacks.	Imprecise by design. Application developer intent is not directly taken into account. Require training phase (clean environment and functionality coverage issues). May be subject to malicious retraining.
Policy	Easy to implement. Low false–positives rate. May be applicable for applications containing intentionally malicious code (backdoors).	Application developer intent is not considered. Vulnerable to 0-day attacks. Requires rule set maintenance. May be ineffective if an attacker is aware of the exact rule set deployed.
Intent	Application developer intent explicitly considered.	Not applicable for applications containing intentionally malicious code (backdoors).
Inputs	Source code not required. Easy to implement. Programming language independent.	May not reveal the actual problem in the application. Selection of inputs to monitor not formalised. Application behaviour ignored.
Application	Allows detecting and fixing bugs in the application itself.	Source code required. Programming language specific.
Outputs	Source code not required. Easy to implement. Programming language independent.	May not reveal actual problems in the application. Selection of outputs to monitor not formalised.

Decisions made based on the developer intent only rely on the fixed implementation. Anomaly-detection based protections are commonly probabilistic and are significantly different in nature as such protections rely on the observed application behaviour to make decisions (Kruegel and Vigna, 2003; Mutz et al., 2006). However, the observed application behaviour is directly affected by the behaviour exhibited by the application users. In other words, anomaly-detection based protection decisions depend on how the users use the application rather than on how the application itself is designed. While focusing on user actions does indeed allow one to detect some anomalies, such anomaly detection does not allow one to determine whether the observed application behaviour is as expected.

In contrast, relying only on the developer intent obviously means that such an approach would not be effective in case the protected application is intentionally dangerous. Consider the application that has a backdoor to allow an attacker to perform arbitrary operations without signing in. None of the attacks performed with the use of the backdoor will be detected, as the observed behaviour can be assumed to be intended. Therefore, using such a protection approach for third-party applications may be essentially a self-deception. However, the same problem is applicable to anomaly detection based protections if an attacker uses the backdoor frequently enough (including the training phase) for the protection to consider the observed behaviour to be usual. Such a limitation naturally narrows the applicability of the approach to in-house solutions or applications with trusted code base. Table 3 provides a summary of conceptual advantages and disadvantages inherent to the proposed classification properties.

In summary, expanding intent-based decisions to consider more intention aspects looks beneficial as such approaches would be able to verify more application behaviour aspects at run time. Detecting unexpected situations (presumably caused by attacks) can also provide enough details to aid in fixing the vulnerabilities.

Perhaps, while much more time-consuming; however, a more general and long-term solution would be to redesign the currently existing web development standards to include such explicit data and code separation. Such significant paradigm shift may indeed help developing more secure web applications; however, the vast amount of currently existing legacy software makes such solution economically infeasible in the nearest future.

References

Ars Technica. Just-released WordPress 0day makes it easy to hijack millions of websites. ⟨http://arstechnica.com/security/2015/04/just-released-wordpress-0day-makes-it-easy-to-hijack-millions-of-websites/⟩; 2015a [retrieved on 20.06.15].

Ars Technica. Serious bug in fully patched Internet Explorer puts user credentials at risk. ⟨http://arstechnica.com/security/2015/02/serious-bug-in-fully-patched-internet-explorer-puts-user-credentials-at-risk/⟩; 2015b [retrieved on 20.06.15].

Antoniol Giuliano, Di Penta Massimiliano, Zazzara Michele. Understanding web applications through dynamic analysis. In: Proceedings of the 12th IEEE international workshop on program comprehension (IWPC'04); 2004.

Artzi Shay, Kiezun Adam, Dolby Julian, Tip Frank, Dig Danny, Paradkar Amit, Ernst Michael D. Finding bugs in dynamic web applications. In: Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA'08); 2008.

Asghar Sandu Usman, Haider Sajjad, Naseer Salman, Ateeb Obaid Ullad. A study of the novel approaches used in intrusion detection and prevention systems. Int J Inf Educ Technol 2011;1(5).

Bezemer Cor-Paul, Mesbah Ali, van Deursen Arie. Automated security testing of web widget interactions. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE'09); 2009.

Bates Daniel, Barth Adam, Jackson Collin. Regular expressions considered harmful in client-side XSS filters. In: Proceedings of the 19th international conference on World wide web (WWW'10); 2010.

Balzarotti Davide, Cova Marco, Felmetsger Vika, Jovanovic Nenad, Kirda Engin, Kruegel Christopher, Vigna Giovanni. Saner: composing static and dynamic analysis to validate sanitization in web applications. In: Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland'08); 2008.

Balzarotti Davide, Cova Marco, Felmetsger Viktoria V, Vigna Giovanni. Multi-module vulnerability analysis of web-based applications. In: Proceedings of the 14th ACM conference on computer and communications security (CCS'07); 2007.

Batory Don, Lofaso Bernie, Smaragdakis Yannis. JTS: tools for implementing domain-specific languages. In: Proceedings of the 5th international conference on software reuse (ICSR'98); 1998.

Buehrer Gregory, Weide Bruce W, Sivilotti Paolo AG. Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th international workshop on software engineering and middleware (SEM'05); 2005.

Balduzzi Marco, Torrano Gimenez Carmen, Balzarotti Davide, Kirda Engin. Automated discovery of parameter pollution vulnerabilities in web applications. In: Proceedings of the 18th annual network and distributed system security symposium; 2011.

Bravenboer Martin, Dolstra Eelco, Visser Eelco. Preventing injection attacks with syntax embeddings. In: Proceedings of the 6th international conference on generative programming and component engineering (GPCE'07); 2007.

Bisht Prithvi, Madhusudan P, Venkatakrishnan VN. CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans Inf Syst Secur 2010;13:2 Article 14.

Bisht Prithvi, Hinrichs Timothy, Skrupsky Nazari, Venkatakrishnan VN. WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of the 18th ACM conference on computer and communications security (CCS'11); 2011.

Boyd Stephen W, Keromytis Angelos D. SQLrand: preventing SQL injection attacks. In: Proceedings of the 2nd applied cryptography and network security conference (ACNS); 2004.

Bai Yuebin, Kobayashi Hidetsune. Intrusion detection systems: technology and development. In: Proceedings 17th international conference on advanced information networking and applications (AINA'2003); 2003.

Chess Brian, West Jacob. Dynamic taint propagation: finding vulnerabilities without attacking. Inf Secur Tech Rep 2008;13(1).

CWE. CWE/SANS Top 25 Most Dangerous Software Errors. ⟨http://cwe.mitre.org/top25/⟩; 2011 [retrieved on 06.20.15].

Chin Erika, Wagner David. Efficient character-level taint tracking for Java. In: Proceedings of the 2009 ACM workshop on secure web services (SWS'09); 2009.

Chang Jian, Venkatasubramanian Krishna K, West Andrew G, Lee Insup. Analyzing and defending against web-based malware. ACM Comput Surv 2013;45(4) Article 49.

Cova Marco, Balzarotti Davide, Felmetsger Viktoria, Vigna Giovanni. Swaddler: an approach for the anomaly-based detection of state violations in web applications. In: Proceedings of the 10th international symposium on recent advances in intrusion detection (RAID'07); 2007.

Chong Stephen, Vikram K, Myers Andrew C. SIF: enforcing confidentiality and integrity in web applications. In: Proceedings of 16th USENIX security symposium on usenix security symposium (SS'07); 2007.

Chandola Varun, Banerjee Arindam, Kumar Vipin. Anomaly detection: a survey. ACM Comput Surv 2009;41(3) Article 15.

Doupé Adam, Cavedon Ludovico, Kruegel Christopher, Vigna Giovanni. Enemy of the state: a state-aware black-box web vulnerability scanner. In: Proceedings of the 21st USENIX conference on security symposium (Security'12); 2012.

Doupé Adam, Cui Weidong, Jakubowski Mariusz H, Peinado Marcus, Kruegel Christopher, Vigna Giovanni. deDacota: toward preventing server-side XSS via automatic code and data separation. In: Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS'13); 2013.

Diebold Boris, Kaufmann Michael. Usage-based visualization of web localities. In: Proceedings of the Asia-Pacific symposium on information visualisation-Volume 9 (APVis'01). Darlinghurst, Australia: Australian Computer Society Inc.; 2001.

Dharmapurikar Sarang, Krishnamurthy Praveen, Sproull Todd S, Lockwood John W. Deep packet inspection using parallel bloom filters. IEEE Micro 2004;24:1.

Du Wenliang, Jayaraman Karthick, Tan Xi, Luo Tongbo, Chapin Steve. Position paper: why are there so many vulnerabilities in web applications? In: Proceedings of the workshop on new security paradigms workshop (NSPW'11); 2011.

Forbes. Wordpress under attack: how to avoid the coming botnet. ⟨http://www.forbes.com/sites/anthonykosner/2013/04/13/wordpress-under-attack-how-to-avoid-the-coming-botnet/⟩; 2013 [retrieved on 06.20.15].

Furr Michael, Foster Jeffrey S. Checking type safety of foreign function calls. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'05); 2005.

Felmetsger Viktoria, Cavedon Ludovico, Kruegel Christopher, Vigna Giovanni. Toward automated detection of logic vulnerabilities in web applications. In: Proceedings of the 19th USENIX Security symposium (USENIX'10); 2010.

Guha Arjun, Krishnamurthi Shriram, Jim Trevor. Using static analysis for Ajax intrusion detection. In: Proceedings of the 18th international conference on world wide web (WWW'09); 2009.

Grechanik Mark, Cook William R, Batory Don, Lieberherr Karl J. Static checking of interoperating components (ICSE'2006); 2006.

Hassan Ahmed E, Holt Richard C. Architecture recovery of web applications. In: Proceedings of the 24th international conference on software engineering (ICSE '02); 2002.

Heiderich Mario, Frosch Tilman, Jensen Meiko, Holz Thorsten. Crouching tiger-hidden payload: security risks of scalable vectors graphics. In: Proceedings of the 18th ACM conference on computer and communications security (CCS'11); 2011.

Hossein Manshaei Mohammad, Zhu Quanyan, Alpcan Tansu, Bacşar Tamer, Hubaux Jean-Pierre. Game theory meets network security and privacy. ACM Comput Surv 2013;45(3) Article 25.

Hooimeijer Pieter, Livshits Benjamin, Molnar David, Saxena Prateek, Veanes Margus. Fast and precise sanitizer analysis with BEK. In: Proceedings of the 20th USENIX conference on security (SEC'11); 2011.

Han Sang-Jun, Cho Sung-Bae. Evolutionary neural networks for anomaly detection based on the behavior of a program. IEEE Trans Syst 2006;36(3):559–670.

Huang Shan Shan, Zook David, Smaragdakis Yannis. Domain-specific languages and program generation with meta-AspectJ. ACM Trans Softw Eng Methodol 2008;18 (Article 6).

Haldar Vivek, Chandra Deepak, Franz Michael. Dynamic taint propagation for java. In: Proceedings of the 21st annual computer security applications conference (ACSAC'05); 2005.

Halfond William GJ, Orso Alessandro. Preventing SQL injection attacks using AMNESIA. In: Proceedings of the 28th international conference on software engineering (ICSE'06); 2006.

Halfond William GJ, Orso Alessandro, Manolios Panagiotis. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In: Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering (SIGSOFT'06/FSE-14); 2006.

Halfond William GJ, Anand Saswat, Orso Alessandro. Precise interface identification to improve testing and analysis of web applications. In: Proceedings of the eighteenth international symposium on software testing and analysis (ISSTA'09); 2009.

Huang Yao-Wen, Yu Fang, Hang Christian, Tsai Chung-Hung, Lee Der-Tsai, Kuo Sy-Yen. Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on world wide web (WWW'04); 2004.

Imran Ghafoor, Jattala Imran, Durrani, Shakeel, Muhammad Tahir. Analysis of OpenSSL heartbleed vulnerability for embedded systems. In: Proceedings of the IEEE 17th international multi-topic conference (INMIC'2014); 2014.

Jovanovic Nenad, Kirda Engin, Kruegel Christopher. Preventing cross site request forgery attacks. In: Proceedings of the 2nd international conference on security and privacy in communication networks (SecureComm'06); 2006a.

Jovanovic Nenad, Kirda Engin, Kruegel Christopher. Pixy: a static analysis tool for detecting web application vulnerabilities. In: Proceedings of the 27th IEEE symposium on security and privacy (Oakland'06); 2006b.

Jovanovic Nenad, Kirda Engin, Kruegel Christopher. Precise Alias Analysis for Syntactic Detection of Web Application Vulnerabilities. In: Proceedings of the ACM SIGPLAN workshop on programming languages and analysis for security; 2006c.

Kruegel Christopher, Vigna Giovanni. Anomaly detection of web-based attacks. In: Proceedings of the 10th ACM conference on computer and communication security (CCS'03); 2003.

Kemalis Konstantinos, Tzouramanis Theodores. SQL-IDS: a specification-based approach for SQL-injection detection. In: Proceedings of the 2008 ACM symposium on applied computing (SAC'08); 2008.

Kukielka Przemyslaw, Kotulski Zbigniew. Analysis of different architectures of neural networks for application in intrusion detection systems. In: Proceedings of the international multiconference on computer science and information technology; 2008.

Liu Anyi, Yuan Yi, Wijesekera Duminda, Stavrou Angelos. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In: Proceedings of the ACM symposium on applied computing (SAC'09); 2009.

Livshits Benjamin, Chong Stephen. Towards fully automatic placement of security sanitizers and declassifiers. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'13); 2013.

Livshits Benjamin, Nori Aditya V, Rajamani Sriram K, Banerjee Anindya. Merlin: specification inference for explicit information flow problems SIGPLAN Not 44, 6; 2009.

Li Jian, Zhang Guo-Yin, Gu Guo-chang. The research and implementation of intelligent intrusion detection system based on artificial neural network. In: Proceedings of international conference on machine learning and cybernetics, volume 5; 2004.

Louw Mike Ter, Venkatakrishnan VN. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In: Proceedings of the 30th IEEE symposium on security and privacy (SP'09); 2009.

Livshits V. Benjamin, Lam Monica S. Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th conference on USENIX security symposium-volume 14 (SSYM'05); 2005.

Lee Wenke, Stolfo Salvatore J. A framework for constructing features and models for intrusion detection systems. ACM Trans Inf Syst Secur 2000;3(4):227–61.

Lee Wenke, Stolfo SJ, Chan PK, Eskin E, Wei Fan, Miller M, Hershkop S, Junxin Zhang. Real time data mining-based intrusion detection. In: Proceedings of DARPA information survivability conference & exposition II (DISCEX'01); 2001.

Li Xiaowei, Xue Yuan. BLOCK: A black-box approach for detection of state violation attacks towards web applications. In: Proceedings of the 27th annual computer security applications conference (ACSAC'11); 2011.

Li Xiaowei, Xue Yuan. A survey on server-side approaches to securing web applications. ACM Comput Surv 2014;46(4) (Article 54).

Musciano Chuck, Kennedy Bill. HTML & XHTML: the definitive guide. 4th edition. O'Reilly Media; 2000 Print ISBN: 978-0-596-00026-4, ISBN 10: 0-596-00026-X.

Mutz Darren, Valeur Fredrik, Vigna Giovanni, Kruegel Christopher. Anomalous system call detection. ACM Trans Inf Syst Secur 2006;9(1):61–93.

Merlo Ettore, Letarte Dominic, Antoniol Giuliano. Automated Protection of PHP Applications Against SQL-injection Attacks. In: Proceedings of the 11th European conference on software maintenance and reengineering (CSMR'07); 2007.

Maggi Federico, Robertson William, Kruegel Christopher, Vigna Giovanni. Protecting a moving target: addressing web application concept drift. In: Proceedings of the 12th international symposium on recent advances in intrusion detection (RAID'09); 2009.

Marco-Gisbert Hector, Ripoll Ismael. On the effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows. In: Proceedings of the IEEE 13th international symposium on network computing and applications (NCA); 2014.

Medeiros Ibéria, Neves Nuno F, Correia Miguel. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In: Proceedings of the 23rd international conference on world wide web (WWW'14); 2014.

Mason Joshua, Small Sam, Monrose Fabian, MacManus Greg. English shellcode. In: Proceedings of the 16th ACM conference on computer and communications security (CCS'09); 2009.

Moonen Leon. Generating robust parsers using island grammars. In: Proceedings of the eighth working conference on reverse engineering (WCRE'01); 2001.

Martin Micahel, Livshits Benjamin, Lam Monica S. SecuriFly: runtime vulnerability protection for Web applications. San Diego, California, USA: Stanford University; 2006 (OOPSLA'05), October 16–20, 2005.

Mitchell Robert, Chen Ing-Ray. A survey of intrusion detection techniques for cyber-physical systems. ACM Comput Surv 2014;46:4 Article 55.

Minamide Yasuhiko. Static approximation of dynamically generated Web pages. In: Proceedings of the 14th international conference on world wide web (WWW'05); 2005.

Nguyen-tuong Anh, Guarnieri Salvatore, Greene Doug, Shirley Jeff, Evans David. Automatically hardening web applications using precise tainting. In: Proceedings of the 20th IFIP international information security conference; 2005.

Nadji Yacin, Saxena Prateek, Song Dawn. Document structure integrity: a robust basis for cross-site scripting defense. In NDSS'09; 2009.

Pietraszek Tadeusz, Berghe Chris Vanden. Defending against injection attacks through context-sensitive string evaluation. In: Proceedings of the 8th international symposium on recent advances in intrusion detection (RAID'05); 2005.

Park Yongsu, Lee Jaeheung, Cho Yookun. Intrusion detection using noisy training data. Computational science and its applications (ICCSA'2004); 2004.

Ray Donald, Ligatti Jay. Defining code-injection attacks. SIGPLAN Not 47, 1; 2012.

Robertson William, Vigna Giovanni. Static enforcement of web application integrity through strong typing. In: Proceedings of the 18th conference on USENIX security symposium (SSYM'09); 2009.

Sung Andrew H, Mukkamala Srinivas. Identifying important features for intrusion detection using support vector machines and neural networks. In: Proceedings of the 2003 symposium on applications and the internet (SAINT'03); 2003.

Scott David, Sharp Richard. Abstracting application-level web security. In: Proceedings of the 11th international conference on world wide web (WWW'02); 2002.

Sun Fangqi, Xu Liang, Su Zhendong. Static detection of access control vulnerabilities in web applications. In: Proceedings of the 20th USENIX security symposium (USENIX'11); 2011.

Schneider Fred B. Enforceable security policies. ACM Trans Inf Syst Secur 2000;3 (1):30–50.

Schmitt Isabell, Schinzel Sebastian. WAFFle: fingerprinting filter rules of web application firewalls. In: Proceedings of the 6th USENIX conference on offensive technologies (WOOT'12); 2012.

Skaruz Jaroslaw. Intrusion detection in web applications: evolutionary approach. In: Proceedings of the international multiconference on computer science and information technology; 2009.

Shum Jimmy, Malki Heidar A. Network intrusion detection system using neural networks. In: Proceedings of the fourth international conference on natural computation volume 5 (ICNC'08); 2008.

Samuel Mike, Saxena Prateek, Song Dawn. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In: Proceedings of the 18th ACM conference on computer and communications security (CCS'11); 2011.

Skrupsky Nazari, Bisht Prithvi, Hinrichs Timothy, Venkatakrishnan VN, Zuck Lenore. TamperProof: a server-agnostic defense for parameter tampering attacks on web applications. In: Proceedings of the third ACM conference on data and application security and privacy (CODASPY'13); 2013.

Saxena Prateek, Molnar David, Livshits Benjamin. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In: Proceedings of the 18th ACM conference on computer and communications security (CCS'11); 2011.

Sommer Robin, Paxson Vern. Outside the closed world: on using machine learning for network intrusion detection. In: Proceedings of the IEEE symposium on security and privacy (SP); 2010.

Son Sooel, McKinley Kathryn S, Shmatikov Vitaly. Diglossia: detecting code injection attacks with precision and efficiency. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security (CCS'13); 2013.

Scholte Theodoor, Robertson William, Balzarotti Davide, Kirda Engin. An empirical analysis of input validation mechanisms in web applications and languages. In: Proceedings of the 27th annual ACM symposium on applied computing (SAC'12); 2012.

Srivastava Varun, Bond Michael D, McKinley Kathryn S, Shmatikov Vitaly. A security policy oracle: detecting security holes using multiple API implementations SIGPLAN Not46, 6; 2011.

Su Zhendong, Wassermann Gary. The essence of command injection attacks in web applications. In: Proceedings of the conference record of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'06); 2006.

Tripp Omer, Pistoia Marco, Fink Stephen J, Sridharan Manu, Weisman, Omri TAJ: effective taint analysis of web applications SIGPLAN Not 44; 2009, p. 6.

Valeur Fredrik, Mutz Darren, Vigna Giovanni. A learning-based approach to the detection of SQL attacks. In: Proceedings of the conference on detection of Intrusions and malware and vulnerability assessment (DIMVA'05); 2005.

Van Gundy Matthew, Chen Hao. Noncespaces: using randomization to enforce information flow tracking and Thwart XSS Attacks. In: Proceedings of the 16th annual network and distributed system security symposium (NDSS'09); 2009.

Wassermann Garry, Su Zhendong. An analysis framework for security in web applications. In: Proceedings of the FSE workshop on specification and verification of component-based systems (SAVCBS'2004); 2004.

Wassermann Garry, Su Zhendong. Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 30th international conference on software engineering (ICSE'08); 2008.

Weinberger Joel, Barth Adam, Song Dawn. Towards client-side HTML security policies. In: Proceedings of the 6th USENIX conference on hot topics in security (HotSec'11); 2011a.

Weinberger Joel, Saxena Prateek, Akhawe Devdatta, Finifter Matthew, Shin Richard, Song Dawn. A systematic analysis of XSS sanitization in web application frameworks. In: Proceedings of the 16th European conference on research in computer security (ESORICS'11); 2011b.

Wired. Massive Attack: Half a million microsoft-powered sites hit with SQL injection. 〈http://www.wired.com/2008/04/massive_attack_half_a_million_microsoft-powered_sites_hit_with_sql_injection/〉; 2008 [retrieved on 20.06.15].

Wired. 8 out of 10 software apps fail security test. 〈http://www.wired.com/2011/12/veracode-report/〉; 2011 [retrieved on 20.06.15].

Wired. Black Hat is Over, But SQL Injection Attacks Persist. 〈http://www.wired.com/2012/08/black-hat-sql-injection/〉; 2012 [retrieved on 20.06.15.].

World Wide Web Consortium. HTML5 specification. W3C. 〈http://www.w3.org/TR/html5/embedded-content-0.html〉; 2014 [retrieved on 17.05.15.].

Xie Yichen, Aiken Alex. Static detection of security vulnerabilities in scripting languages. In: Proceedings of the 15th conference on USENIX security symposium-Volume 15 (USENIX-SS'06); 2006.

Yu Fang, Alkhalaf Muath, Bultan Tevfik. STRANGER: an automata-based string analysis tool for PHP. In: Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems (TACAS'10); 2010.

Zheng Yunhui, Zhang Xiangyu. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In: Proceedings of the 2013 international conference on software engineering (ICSE'13); 2013.

Zheng Yunhui, Zhang Xiangyu. Static detection of resource contention problems in server-side scripts (ICSE'12); 2012.