1. What is the paper about? What is/are the vulnerability? What causes the vulnerability?

The paper discusses a new defense strategy against cross-site scripting (XSS) attacks called Blueprint. According to the paper, the number one security threats for the Internet at large are XSS attacks. The attacks rely upon unauthorized script code injected into a web page to extract confidential and sensitive user data. The malicious script code is embedded in untrusted HTML and is executed on the victim’s web browser within the context of the vessel website. Enforcing a no-script policy for untrusted HMTL to prevent unauthorized script execution is the aim of defenses against XSS attacks. The strategy of the defense outlined in this paper and the goal for Blueprint is to minimize the burden on the browser in determining untrusted content.

1. What is/are the contributions of the paper? How was the vulnerability or insecurity discovered?

The paper discusses two main defense approaches: content filtering and browser collaboration. In content filtering application attempt to detect and remove all scripts from untrusted HTML while browser collaboration gets information from the browser as to which scripts are authorized.

Content filtering, also widely known as sanitization, uses specially designed filter functions on user input to protect against/remove any potentially malicious data or instruction. Simple sanitization, such as disallowing HTML syntax control characters, because every control character that can be used in a malicious attack code/script also has a legitimate use in many legitimate non-script contexts. Advanced content filters look at untrusted content and try to predict how the client web browser’s parser will interpret it. This not only makes the web application’s job of protection depend in part on the different web browser parsers, but also leaves the content filters open to exploit via any anomalous behavior by any of the browser parsers.

Browser collaboration involved tools to help browsers distinguishing authorized scripts from unauthorized scripts. Protocols are used to communicate a set of authorized/unauthorized scripts, which the browsers use to enforce policies denying execution of the unauthorized scripts. This unfortunately requires web application to use custom browsers to be able to communicate with the server containing the set of scripts; therefore, it lends itself more to a long-term solution, where all browsers already contain this communication by default, rather than a near-term one. Also, the standards by which all browser use for communication would have to be agreed upon, which in itself is a long arduous process.

1. The detailed techniques to solve the problem.

Blueprint claims to satisfy the three main objectives of preventing XSS attacks: 1) it is robust, protecting even with browser quirks, 2) it supports structure, benign HTML derived from untrusted user input, and 3) it is compatible with existing browsers current in use. Since the parsing behavior of browser can be unreliable, Blueprint effectively takes over and controls the parsing decisions instead of the browser. The Blueprint application uses information about the flow of untrusted HTML in a browser to create a “blueprint” or structural representation of untrusted web content, with XSS attacks removed. The implementation avoids understanding how the browser parses the data and instead enforces the application’s understanding of the web content on the browser. In doing this, the effect of the browser parser’s analogous behavior does not come into play.

By systematically reasoning about the flow of untrusted HTML in a browser, we develop an approach that provides facilities for a web application to automatically create a structural representation — a “blueprint” — of untrusted web content that is free of XSS attacks. Our approach employs techniques to carefully transport and reproduce this blueprint in the browser exactly as intended by the web application, despite anomalous browser parsing behavior. Our general approach offers strong protection against script injections, and enables support for complex script-free HTML user input. Extensive experiments with BLUEPRINT demonstrate its resilience against subtle XSS attacks, reasonable performance overheads, compatibility and effectiveness on over 96% of existing browser market share.

responds/reacts to the attacks. Static Code Checkers can be used to detect one of the main causes of vulnerabilities, improper type checking of input. Combined Static and Dynamic Analysis is used of normal SQL keywords, with a proxy filter to de-randomize the keywords before they are applied to the database.

Main idea. The crux of our approach is to eliminate any dependence on the browser’s parser for building untrusted HTML parse trees. That is, we eliminate the use of path *B* and instead derive an alternative path to render untrusted content without the risk of XSS attacks. In our approach, the following steps are performed by the web application:

1) On the application server, a parse tree is generated from untrusted HTML with precautions taken to ensure the absence of dynamic content (e.g., script) nodes in the tree.

2) On the client browser, the generated parse tree is conveyed to the browser’s document generator without taking vulnerable paths such as *B* which involve unreliable browser parsing behavior. This two-step process ensures untrusted content generated by the browser is consistent with the web application’s understanding of the content. The generated document reflects the application’s intention that the untrusted content does not contain scripts, therefore all unauthorized script execution is prevented.

In our approach, we create the parse tree for untrusted content programmatically using a small set of low-level Document Object Model (DOM) primitives that are well documented [6] and supported on all JavaScript-enabled browsers. Input is provided to these DOM APIs (via path *E* ) as both instructions and data: instructions define parse tree structure and node types, and data (e.g., character data in text nodes) annotates nodes of this tree. Once these explicit instructions are given, the browser’s DOM implementation constructs the untrusted HTML parse tree, then supplies this parse tree to the document generator through the final transition *R*. Our goal therefore reduces to reliably transporting both instructions and data safely to path *E* for invoking DOM APIs.

Transport of instructions. Transporting instructions is relatively easy: we simply have the web application generate trusted JavaScript code that flows through paths (*A, B, C, D, E*). Since instructions are entirely devoid of untrusted content, we can ensure they will be correctly delivered to path *E*. This claim in supported by the discussion oat the start of the section about obtaining predictable behavior from trusted content through testing on various browsers.

Strictly interfacing with the DOM via trusted code is not enough to ensure the web application’s intended parse tree ultimately prevails. It is still possible for trusted code to introduce script nodes into the parse tree, perhaps unintentionally if the code uses certain DOM API methods that trigger parsing activity. For instance, by invoking *document.write()* (path *F*), character data may be explicitly supplied to the HTML parser, and may then violate structural integrity of the intended parse tree or even the structure of trusted HTML content. There are many similar interfaces that trigger unsafe parsing behavior, such as the eval() function (path J ), and the innerHTML property (path *F*). We take care that our trusted client-side code does not use these unsafe APIs as it constructs parse trees.

Reliable transport of untrusted data. The final component of our solution is safe transport of untrusted data from path *A* to *E* for providing input to DOM APIs. If raw data is exposed to the browser’s parsers, it is practically infeasible to guarantee all XSS attacks are prevented. This is because raw data might contain control character sequences uncaught by a server-side filter, and these characters can cause the formation of script nodes if the browser’s parser interprets them as such. Furthermore, untrusted data must be effectively isolated from trusted code/data to preserve integrity of trusted content.

To address the threat posed by raw untrusted data, we convert this data to an encoded representation that consists only of characters from a strictly defined “safe alphabet”. This alphabet (say “a–z”) only contains characters that are syntactically inert; that is, none of the safe alphabet characters cause changes in the syntax state of the browser’s HTML parser as they are processed. Therefore, for all possible sequences of safe alphabet characters the browser’s parsing behavior can be reliably anticipated.

As they are safe to use for data transport without affecting document structure, we expose these encoded characters in a text node to the browser’s HTML parser (shown in Figure 1 taking the alternate route *B’*).

We then take steps to ensure that untrusted data completely bypasses the browser’s unreliable Java-Script parser. Instructions supplied over path *E* extract the encoded text node from the DOM via paths (*Q, P*), thus avoiding the JavaScript parser altogether. We decode and recover the raw untrusted character data to the JavaScript runtime environment’s memory state, then supply it to DOM APIs as input via path *E*. In summary, untrusted data traverses through path (*A, B’, Q, P, E*) in the figure.

We now can say the processing of raw untrusted data by the browser’s DOM implementation will not result in unauthorized script execution because (as described above) we only employ DOM APIs that do not trigger parsing behavior. The parse tree ultimately generated using our approach is thus supplied via *R* to the document generator, which successfully renders the document free of any XSS attacks.

1. What are the strength/weaknesses of the paper?

The first strength of the paper is the clarity in which they discuss the many different injection mechanism as well as the attack intents. I feel it is very important to know and understand both the way in which the attackers can exploit vulnerabilities, but also what they intended to gain from their attack. By looking at what the attackers were after that will help the developers put extra scrutiny on those items that may give attackers access to the information they are after and protect any avenues for them to obtain it. Another strength is they assessed different tools for the prevention/protection not only for how well they did their intended job, but whether or not base code needed to be modified, whether detection/prevention were automated, and identified additional infrastructure that might be necessary.

The first weakness I could see with the paper was they evaluated the techniques based on certain tools, but did not really discuss the effectiveness of the overall technique in general. One technique may be highly effective when used appropriately, but no tools may yet exist to provide an effective means to apply the technique. A second weakness I would identify is they did not spend as much time talking about defensive coding practices even though use of these practices address the root cause of most SQL injection vulnerabilities and is the most straightforward solution.

1. What can you do better?

The authors of the paper compiled a lot of very useful information about SQL injection attacks and the current techniques for detecting and preventing them. They were also able to discover trends in how many existing products/techniques fail to detect/prevent attacks. However, the authors fail to provide their own way to combat these trends or suggestions as to how to prevent against these pitfalls when creating a product to detect and prevent SQL injection attacks. In addition, the authors also fail to address any possible future attacks. For example, their research could have shown areas which were vulnerable to attacks, but that have not yet been exploited or do not yet have a technique to provide protection against them. This would not only help them in creating a new product/technique for the industry, but also helping those who wish to improve their own products/techniques to try to get ahead of the curve.