Large-Scale, Automatic XSS Detection using Google Dorks

Riccardo Pelizzi *Tung Tran* *Alireza Saberi*

Abstract

XSS Attacks continue to be prevalent today, not only because XSS sanitization is a hard problem in rich-formatting contexts, but also because there are so many potential avenues and so many uneducated developers who forget to sanitize reflected content altogether.

In this paper, we present Gd0rk, a tool which employs Google's advanced search capabilities to scan for websites vulnerable to XSS. It automatically generates and maintains a database of parameters to search, and uses heuristics to prioritize scanning hosts which are more likely to be vulnerable. Gd0rk includes a high-throughput XSS scanner which reverse engineers and approximates XSS filters using a limited number of web requests and generates working exploits using HTML and JavaScript context-aware rules.

The output produced by the tool is not only a remarkably vast database of vulnerable websites along with working XSS exploits, but also a more compact representation of the list in the form of google search terms, whose effectiveness has been tested during the search.

After running for a month, Gd0rk was able to identify more than 200.000 vulnerable pages. The results show that even without significant network capabilities, a large-scale scan for vulnerable websites can be conducted effectively.

1 Introduction

Cross-site scripting (XSS) has emerged as one of the most serious threats on the Web. CWE/SANS Top-25 [35] lists XSS first in its list of "Top 25 Most Dangerous Software Errors", while the web-application focused OWASP lists XSS second in its list of top-10 security risks [36]. In terms of raw numbers, it is the most commonly reported vulnerability over the past year, accounting for 12.8% of all reported CVE vulnerabilities. (See Figure 1).

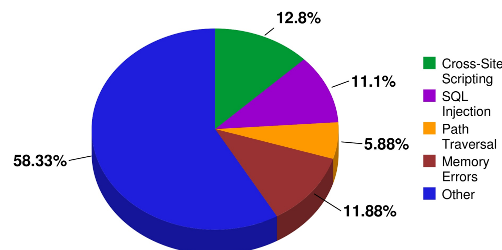

Figure 1: CVE vulnerabilities for 2010

The increase in prevalence and severity of XSS attacks has spawned several research efforts into XSS defenses. Many [19, 34, 23, 37, 18, 38, 4, 31] of these efforts have focused on the server-side, and attempt to detect or prevent unauthorized scripts from being included in the server output. Several researchers [18, 23, 37, 34] have eloquently argued that no server-side logic can accurately account for all "browser quirks", undocumented or obscure HTML parsing tricks that can be used to bypass filters. Also, DOM-Based attacks cannot be detect by server-side filters, which can only analyze an HTML document as a static entity. As a result, hybrid approaches have been developed that combine varying degrees of client-side support with a primarily server-side XSS defense [37, 23, 31]. However, the diffusion of such methods remains limited: hybrid methods require support from both clients and servers. Since the party that is most directly affected by an XSS attack is the user who accesses a vulnerable server, client-side protections are thus desirable (despite their limitation to so-called *reflected XSS*) and have been developed [29, 4, 17, 11]. However, client-side defenses are no definitive solution either: IE8 regular-expression based approach is easy to circumvent [24] and has been criticized for opening new bugs in non-vulnerable web applications [25]; Chrome's XSSAuditor has been merged into mainline Webkit more than a year ago, but was only recently enabled by default due to compatibility concerns [3]. Moreover, it does not protect against all XSS attack vectors, but only against those attacks that inject a complete, self-contained script; although favored by experienced users, NoScript requires too much thinkering for the average browser user.

For these reasons, XSS prevention is still largely left in the hands of server-side sanitization functions in web applications. Unfortunately, XSS sanitization is a hard problem: not only it is hard to write a filter that blocks XSS attacks and allows rich-text formatting (allowing tags such as `<b>` and `<i>`), but even in simpler cases, developers often forget to sanitize data at all, or use a sanitization routine designed for the wrong context. For example, if a GET parameter is echoed inside a JavaScript string, using a sanitization routine written for plain-text sanitization might not work: if the routine strips angled brackets to prevent injection in the form of `<script>...</script>`, the attacker can break out of the JavaScript string and inject malicious JavaScript code (displayed in red):

```
<script>
var query = "out"; attack();//";
</script>
```

Static analysis tools to verify the correctness of sanitization functions with respect to the content where they

appear do exist [2, 39] but they are not widely employed. Thus, many websites are still vulnerable to XSS attacks, and exploits are for the most part trivial.

Google has already been employed as a tool to scan for web vulnerabilities [10] such XSS [9, 7] and SQL injection [15, 8, 41]. Its advanced features allow to search for specific strings in the text and in the URL of indexed pages. For example, text search can be used to look for specific error messages that reveal useful information about the web application, while URL search can be used to detect multiple deployments of specific web applications that are known to be vulnerable, or to search for web application scripts with a particular behaviour (such as redirection scripts and mailing scripts). Examples can be found in the Google Hacking Database [13]. However, the tools referenced above only search and report vulnerabilities according to a fixed list of search terms. When these terms are effective in detecting vulnerable sites, they are called *google dorks*.

2 Overview of Approach

This paper presents Gd0rk, a tool which employs Google's URL search capabilities to find vulnerable sites and automatically maintains a database of HTTP parameters and script names to heuristically drive the scan towards hosts that are more likely to be vulnerable. The tool includes a high-throughput XSS scanner that evaluates XSS filters using only one web request per parameter and generates working exploits for vulnerable sites using HTML and JavaScript context-aware rules.

To begin the scan, Gd0rk needs to be bootstrapped with a list of parameter names and scripts related to vulnerabilities. This does not need to be an exhaustive list, because Gd0rk is able to find new parameters and scripts. Gd0rk has two main threads: one searches for parameters and scripts on Google, while the other scans Google search results for XSS vulnerabilities. Both create subthreads to perform parallel requests to minimize the delay caused by round trip time and use bandwith efficiently.

The Google Search thread selects either a script or a parameter name and builds a search term from it: the parameter is paired with a popular language extension such as `asp` or `php`. For example, if the parameter `search` and the extension `jsp` are chosen, the Google search `allinurl: search jsp` is launched. Issuing numerous search queries to Google without user interaction requires addressing the following problems:

1. Even though Google offers a search API with bindings for many languages, it is limited to either 25 results per term or 100 queries per day. To solve this, Gd0rk scrapes the results off the Web Interface.
2. The web interface throttles traffic from IPs which generate too many requests. The limit is even lower for advanced searches, such as those containing the `allinurl:` modifier; after this limit has been exceeded, Google presents a CAPTCHA challenge to the user to increase the limit. Since CAPTCHAs can be solved from different IPs and used at a later time, Gd0rk includes a small tool to solve a large number of CAPTCHAs efficiently, to allow searches to run uninterrupted. Moreover, we use a very small number of different IPs and send a limited amount of requests per minute. This does not reduce the scan speed significantly, as one web request to Google can queue up to 100 URLs for the XSS scanner.

The XSS scanner thread selects one of the search results generated from the Google thread according to a heuristic which prioritizes results containing parameters that already appeared in vulnerable websites. The search result URL is used to generated multiple *scan URLs*, one for each URL parameter: the involved parameter value is modified to include a special scan string, whose purpose is to detect where the parameter is reflected in the HTML page and how the string is sanitized against XSS attacks (if it is sanitized at all). For example, the scan URL for the parameter `term` and the google result:

```
http://vuln.com/search.php?term=hello&b=1
```

becomes [1]

```
http://vuln.com/search.php?term=hello#<a>a"a'#&b=1
```

This allows Gd0rk to probe each parameter for XSS vulnerabilities using just one HTTP request, instead of trying one specific concrete attack at a time. This is important because the Google search thread can create up to 100 results for each request; therefore, Gd0rk's throughput largely depends on the throughput of the XSS scanner. The server response to the scan URL is used to create a *translation table*, an approximation of the filter's sanitization behaviour: by looking at how the scan string is reflected in the document, it is possible to detect how single characters are sanitized. A scan string might appear more than once in the document: each reflected instance of the parameter has its own translation table, as they might be different. Each reflected instance is then passed to the XSS exploit generator: this module detects the *context* of the reflection (the location of the reflected scan string in the HTML parse tree) and attempts to build a syntactically correct attack compatible with the sanitization employed by the web application. For example, given a parameter reflected inside a JavaScript double-quoted string:

```
<script>
var query = "param";
</script>
```

the exploit generator devises two different XSS attacks. The first attack breaks out of the string, writes malicious code and then opens another

[1]This is a simplified format for the scan string.

variable assignment to resync with the remaining JavaScript code provided by the web application (";alert(1); var foo = "). The second attack closes the script tag and insert a whole new malicious script (</script><script>alert(1)</script>). Assuming that the web application escapes double quotes ('"'-> '\"'), the first attack is rejected because it is not compatible with the translation table, while the second attack is accepted.

If an exploit is found, the priority of other results from the same parameter search is increased; these become more likely to be selected by the scanning thread. Moreover, all new parameters found in the vulnerable URL are queued for a Google search. This allows Gd0rk to expand its search in new directions and find new potential keywords.

Contributions

To summarize, the paper makes the following contributions:

- A tool to uncover a large number of XSS vulnerabilities and google dorks using Google Search, to demonstrate that attackers do not require substantial network resources to mount a large-scale scan.
- A context-aware high-throughput XSS scanner which probes websites for reflected XSS vulnerabilities with a minimal amount of HTTP requests and produces syntactically working exploits.
- A report about the XSS vulnerabilities collected, including their nature and their status after a year.

Organization

Section 3 briefly introduces XSS attacks; Section 4 describes Google Search advanced capabilities and its throttling policies. Section 5 describes the XSS scanner, including a model of its context-aware rules using an FSA. Finally, Section 6 presents the results collected over the course of a month.

3 XSS Attacks

XSS Attacks are web vulnerabilities that allow an attacker to inject malicious code in a web page served to a victim user. Although the attacker is able to run his code on the user's browser by hosting it on his own malicious website and tricking the user into visiting it, the code runs in a sandbox: the same-origin policy (SOP) enforced by the browser prevents the attacker's code from stealing the user's credentials on any other web site, or observing any (potentially sensitive) data exchanged between other websites and the user. However, XSS attacks allow the attacker to circumvent the SOP, because a vulnerable site will embed the attacker code directly into one of its webpages, that is, within the domain boundaries enforced by the SOP.

Exploiting an XSS vulnerability involves three steps. First, the attacker uses some means to deliver his malicious payload to the vulnerable web-site. Second, this payload is used by the web site during the course of generating a web page sent to the user's browser. If the web site is not XSS-vulnerable, it would either discard the malicious payload, or at least ensure that it does not contribute to JavaScript code content in its output. However, if the site is vulnerable, then, in the third step, the user's browser would end up executing attacker-injected code in the page returned by the web site.

There are two approaches that an attacker can use to accomplish the first step. In a *stored XSS attack,* the injected code first gets stored on the web-site in a file or database, and is subsequently used by the web-site while constructing the victim page. For instance, consider a site that permits its subscribers to post comments. A vulnerability in this site may allow the attacker to post a comment that includes <script> tags. When this page is visited by the user, the attacker's comment, including her script, is included in the page returned to the user.

In a *reflected XSS attack,* the attacker lures the user to click a link or visit a malicious web page, which causes a request from the user to be sent to the vulnerable website. This request will include one or more malicious parameters properly crafted by the attacker. When a vulnerable web site uses these parameters in the construction of a response's HTML parse tree (either because it echoes these parameters into the response page directly without proper sanitization, or because it serves JavaScript code that uses this data dynamically to build DOM nodes on the browser), the attacker's code is able to execute on this response page. When crafting the parameters, the attacker must take care of two elements:

1. The web application might filter or modify the input provided to prevent XSS attacks. This process is called *sanitization*. The attacker needs to work around the sanitization filter to output syntactically correct malicious code.
2. The syntax required to execute malicious code depends on the context where the parameter is echoed in the Web page. For example, if the parameter is supposed to be visible text, then a suitable attack would be in the form of <script>xss();</script>, inserting a new script node. However, if the parameter is echoed inside a JavaScript string, the attack would be in the form of "; xss(); //, breaking out of the string and directly injecting malicious code in the existing script node.

For example, Figure 2 shows how a reflected attack can be carried out on a vulnerable website: maliciously crafted input can open a script node in the middle of the page and execute JavaScript code in the context of the web application. This code will thus have access to the domain cookies, and may send them to an external location controlled by the attacker.

PHP Code

```
<html>
  <head>
    <title>Vulnerable Page</title>
  </head>
  <body>
    <h1>Sorry, 0 search results returned for
    <?php echo \$_GET["term"]; ?></h1>
  </body>
</html>
```

Malicious Input

```
http://a.com/search?term=<script>document.location=
  'http://evil.com/' + document.cookie</script>
```

Figure 2: Reflected XSS Example

4 Google Search

Originally based on the pagerank algorithm [26], indexing billions of web pages, Google Search provides a public, large database of URLs. Normally, Google is used as a keyword-based search engine: users enter search terms and the most pertinent results are returned. However, Google also provides many advanced search options. For example, it allows users to restrict the search to the text content of HTML links, to the page title, or to a specific domain. One interesting option searches for content in the page URL. To activate this option, users must prepend their search terms with `allinurl:`. Searching for a specific URL format might be more helpful than a keyword-based search for certain purposes. For example, `allinurl: forum aspx` searches for forum applications written in ASP.NET. A keyword-based search could easily search for forums, but would not be able to express the language constraint.

Google advanced search features have been used effectively to find security issues. [13] shows many examples of search terms used to find informative error messages, password files, online devices and sensitive services. For example, the search string

```
"Error Diagnostic Information"
intitle:"Error Occurred While"
```

can be used to search for Coldfusion error pages. The very first result on google for this query displays a complete SQL query, along with a short snipped of Coldfusion code. This can be very helpful information for an attacker trying to reverse engineer the database structure to perform a successful SQL injection. These search terms that expose security-related pitfalls are called *google dorks*. Specifically, Google Dorks have been used successfully to build tools that automatically detect XSS [9, 7] and SQL Injection [15, 8, 41] vulnerabilities. This is because XSS attacks (in their reflected form) and SQL Injections vulnerability assessment are relatively simpler to automate using a black-box approach, while other problems uncovered by google dorks can only help a manual attacker to gather helpful information to launch a successful attack.

To automate the construction of search terms, Gd0rk only uses strings in the form of `allinurl: <parameter> <extension>` and `allinurl: <script>.<extension>`; `parameter` is a parameter name as previously found in a URL, `script` is a filename found in a URL and `extension` is taken from the set `[asp,php,jsp,cgi]` to restrict the search to simple, dynamic applications.

4.1 Advanced Search Throttling

Unfortunately, Google does not allow users to easily retrive a large deal of search results programmatically. The JSON API, which is specifically offered for automated querying, is limited to 100 free queries per user per day. Moreover, it requires a valid key from Google. The alternative is to scrape the web interface. Unfortunately, this approaches faces other challenges:

- The most recent version of the Google Search interface is heavily dynamic and presents all its search results to the user through JavaScript DOM manipulation. To scrape the results from this page, it would be necessary to simulate a full-fledged JavaScript engine. Luckily, results in plain HTML are still served to older clients for compatibility reasons. Thus, we perform the query spoofing the user agent and impersonating Internet Explorer 6.
- Google limits the rate searches that can be performed from a single IP; moreover, this limit is even lower for advanced searches, such as those containing the `allinurl:` modifier. After the threshold has been exceeded, Google presents a CAPTCHA challenge to the user. If the user solves the challenge successfully, Google returns a cookie that can be used to perform more queries, exceeding the threshold rate. We discovered that the cookie is not strictly associated to the IP that solved the challenge: the CAPTCHA can be solved from one IP and the cookie can be used at a later time with a different IP. For this reason, Gd0rk includes a small tool (shown in Figure 3) to quickly solve CAPTCHAs and avoid interruptions to the Google crawl. The number of CAPTCHAs required for the search to continue uninterrupted is modest: since one single request returns up to 100 results, the XSS scanner is more likely to be the bottleneck and the Google search thread can proceed at a slower pace. To increase the speed of the search, we use a limited number of IPs. We also experimented with proxies: since Web Proxies offer greater speed and availability than HTTP proxies or TOR [16], we wrote a Python module to send request through web proxies running a popular Web Proxy software, Glype [12]. Unfortunately, it seems that these proxies are either handled differently by Google's throttling policy, or that they

Figure 3: Cookie Generator

already send too many requests to Google from other users. Whatever the reason, we were not able to send many requests before Google blocked us or asked for new CAPTCHAs.

Each parameter can yield up to 4000 URLs: Google serves up to 1000 results for each query (one 100 per page), and every parameter is combined with 4 extensions to yield 4 queries. These URLs are sent to the XSS scanner queue, which is described in the next section.

5 XSS Scanner

Gd0rk includes an automatic black-box XSS vulnerability scanner, which is able to identify XSS vulnerabilities due to incorrect sanitization and generate a working exploit using only a single HTTP request per parameter. This allows Gd0rk to scan a high number of website in a small amount of time, which is critical for a large-scale tool.

The scanner's accuracy in detecting vulnerable pages and generating working exploits depends on its ability to:

- reverse-engineer sanitization functions employed by web applications and approximate them as character transformations.
- detect all occurrences of the reflected parameter in the page and parse the page to understanding their parse tree contexts.

To accomplish both goals, the scanner performs one HTTP request for each parameter in the query string, modifying these in turn by appending a scan string to their values.

Injecting the scan string in all parameters at once would require fewer requests and speed up the scan, but it would also decrease its accuracy because changing all parameters at once would more likely cause an error page to be returned instead of a page constructed with ordinary application logic. Instead, both types of page can contain vulnerabilities and should be tested. The format of the scan string is the following:

```
MARKER SYMBOL1 SEP SYMBOL2 SEP ... MARKER
```

where the `MARKER` couple is used to isolate and detect all occurrences, and `SEP` acts as a separator between `SYMBOL`s. Currently, `MARKER` and `SEP` are alphanumeric identifiers, because the scanner expects to find them unsanitized in the response. Finally, each `SYMBOL` is a character that is probed for sanitization by the web application. By searching for the scan string in the response, it is possible to extract the value of each $f(symbol_i)$, where f is the sanitization function. Thus, a table of `filter input -> filter output` entries can be built, which represents an approximation of the filter. For example, if the scan string is `MARKER < SEP > MARKER` and the string `MARKER < SEP > MARKER` is found in the response, the translation table is `{'<': '<', '>': '>'}`. Any character not probed by the scan string is considered unsanitized by f; thus, all symbols potentially involved in XSS exploits generation are present in the scan string, to avoid optimistic approximations. The scan string also contains common multicharacter tokens such as `<script>` and `document.cookie`, to approximate the filter's behavior towards nontrivial HTML syntax. Note that each distinct reflected instance of the scan string has its own translation table (as these may be different), so that the exploit generation phase can analyze and eventually generate an exploit for each instance.

Once the translation tables have been built, it is necessary to identify the HTML context (and eventually, the JavaScript context) of each instance. The term "context" refers to the type of node in the HTML parse tree where the scan string can be found. This phase is essential to construct a syntactically valid XSS exploit. Before parsing the response, each reflected scan string is replaced with a unique alphanumeric idenfier. This can be done easily using regular expressions since the scan string has known start and end markers. This is important because the scan string contains special HTML and JavaScript characters: if these were left in the response, the parser would need to be modified to discern special characters from special characters in scan strings. Instead, by replacing scan strings with alphanumeric identifiers which are guaranteed not to affect the HTML parse tree, we can use a parser for ordinary HTML code (actually a lexer, built with flex [27]) to infer the HTML context. If the identifier is found inside a script tag, a script URL, an event handler or a javascript URL, Gd0rk uses SpiderMonkey's reflection API [14] to obtain the path from the root of the JavaScript parse tree to the element which contains the reflection. For example, given this simple HTML snippet, where the `@` sign marks the location of the reflected instance:

```
<html><body>
<script>
var query = "@";
</body></html>
```

The HTML context is SCRIPT TAG, while the JavaScript context is ['VarDecl', 'Right-Hand', 'Literal']. The rationale behind the hierarchical format for the JavaScript context is that the exploit generation should not only insert valid code (breaking out of the string in this case), but also do so in the outermost scope, where it is executed under all conditions. Using a hierarchical format for JavaScript, the exploit generation phase can handle arbitrarily complex contexts. HTML injection does not need escaping to the outermost context, as most tags are allowed to have JavaScript code in their descendants. Therefore, it is not important to know the hierarchy of tags from the root node to the scan string.

Finally, after calculating the character translation table and the context, the XSS scanner attempts to produce a working exploit. From the given HTML context, Gd0rk devises an attack string and attempts to construct it through the character filter: for each character in the attack string, Gd0rk looks for a corresponding input character in the translation table. If the character is not present, then it is considered unsanitized and can be used. If the character is present, the exploit generator looks for the input character that translates to the required output character. This allows the filter to handle encodings such as '<'->'<' and select the appropriate character to output <. The rules used by Gd0rk to generate exploits are presented using an augmented FSA, shown in Figure 4. It should be interpreted in the following way:

- The states represent the type of HTML node where the scan string was found. Since we are approximating HTML as a regular language, some state represent non-regular behavior that we do not wish to approximate. For example, text inside a title tag should not be modeled together as text inside a p tag, as the former cannot contain scripts. JavaScript content in tags and attributes is also treated differently, as it presents the additional opportunity of injecting directly within the existing code.
- The initial state is the context as returned by the HTML parser
- The edges represent text that needs to be written through the filter to switch to another state. If the filter does not allow writing such characters, the translation is not permitted.
- The red transitions represent attempts to insert an XSS attack. They are called *attack transitions*.
- A valid path starts from the initial state, includes one attack transition and ends back on the initial state.

The concrete implementation does not actually use an FSA, but is instead a Python function which encodes a set of rules. Moreover, the FSA is actually a simplified representation of the concrete FSA expressed by these rules: firstly, only HTML contexts are represented, abstracting JavaScript context handling through the "To JS" attack transition. This transition actually represents the entry point to a Python subroutine that processes the JavaScript context. This routine first scans the context hierarchy from the element to the root, generating JavaScript code that would close such contexts. Then, it inserts the actual XSS payload. Finally, it scans the context hierarchy from the root to the element to reopen such contexts, to resynch with the remainder of the script generated by the web application. Secondly, some states have been abstracted and merged into a single state: the "Attribute Value" state does not really have transitions to "Attribute" for {',", }. Instead, only one of them is permitted for each context, because the state in the graph represents three distinct states for attributes opened with single quotes, double quotes or no quotes at all respectively.

We present a simple example to illustrate how the paths on the FSA and the JavaScript subroutine yield syntactically correct attacks. Consider the following snippet, where the @ character represents the injection context:

```
<html><body>
<script>
function foo() {
  var query = "@";
  return query;
}
</body></html>
```

The HTML context is "Script Tag". The attacks devised by the FSA are the following:

1. Use the JavaScript subroutine to inject JavaScript code directly.
2. Close the script tag with </script> and use an attack from the "Text" context. Finally, reopen a script tag.

If the sanitization routine escapes angled brackets, the second path would not be accepted because the generator is unable to write </script> through the filter. Therefore, the only valid path is through the JavaScript routine. The JavaScript context is

```
'FunDecl', 'BlockStmt', 'VarDecl, 1, 'Literal'
```

Scanning through the context hierarchy from the element to the root yields the string ";}. Then, the actual payload alert(1) is inserted. Finally, the context is scanned again in reverse to resync the script, yielding function foo() { var bar = ". This generates a syntactically correct exploit that executes as soon as the script tag is evaluated, without having to call the function foo. The red text shows the exploit as prepared by the FSA:

```
<html><body>
<script>
function foo() {
  var query = "";} alert(1); function foo() {
var bar = "%";return query;</body></html>
```

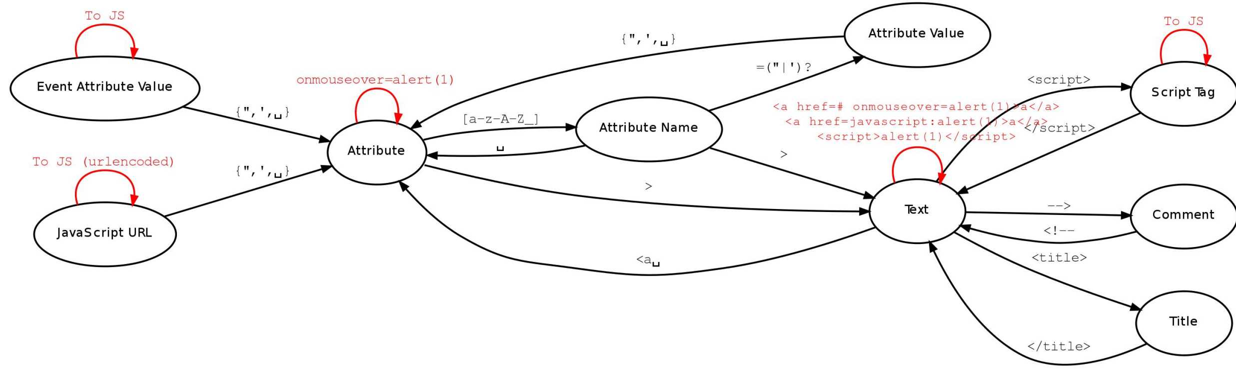

Figure 4: HTML Contexts

On the other hand, if the filter forbids closing the JavaScript string but permits angled brackets, the transition to the "Text" context is valid; this yields `</script>`. From there, the first valid path is taken, yielding `<script>alert(1)</script>`. Finally, the transition back to "Script Tag" yields `<script>`.

If an explot is successfully generated, the exploit string is saved in the database and the search result being scanned is marked as vulnerable. Since the heuristic to select results for the analyzer thread is based on the vulnerable to not vulnerable ratio, this increases the priority of results from the same search to be analyzed in the future. Moreover, all the other parameters present in the URL are queued for a Google search if they have not been searched already. The final result of the scan is a set of responses to scan URLs, one for each querystring parameter; for each response, a vulnerability analysis of all reflected instances is saved on the database, along with a syntactically correct exploit if the instance is vulnerable.

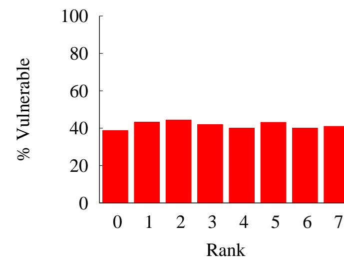

Figure 5: Pages still vulnerable by Pagerank

6 Results

We ran Gd0rk for approximately 30 days during April 2010, using 4 IP addresses; it was bootstrapped with a short list of URL parameters, taken from vulnerable web applications from recent CVE advisories [6]. The tool searched 18275 parameters on Google for a total of 29 million search results. Each request to Google returned 99 results on average, for a total of approximately 300000 requests. When the crawl was suspended, 68807 parameters were still queued for search. It found a total of 272051 vulnerable websites, that is, 0.94% of the total websites scanned. The data contrasts with [22], which reports a 4.30% vulnerability rate. The reduced incidence might be due to web developers being more educated about XSS in 2010 than in 2006, or due to the different type of XSS vulnerability scanned by Secubat (XSS in HTML forms target URLs).

Unfortunately, the scan was done with a preliminary version of Gd0rk that did not feature the context-aware XSS scanner. Rather, the tool simply injected the string `<script>alert(1234)</script>` as a scan string and checked for an exact match in the response. Also, Gd0rk did not search for queries of the form `<scriptname>.<ext>`, but only used parameters-based queries. On the other hand, the data's age allowed us to present additional information regarding how many vulnerabilities are still present after a year. We recently reanalyzed a randomly selected subset of 13295 vulnerable webpages with the new XSS scanner. Our results indicate that 36.32% of the websites are still vulnerable. We manually verified 100 generated exploits and confirmed that 96 of them actually work. Two non-working exploits were parameters echoed inside `frameset` tags, which the scanner does not yet support and treats as `text`. However, `script` tags inside `frameset`s are not executed. This can be fixed by implementing appropriate rules for the context. One error was due to the application logic accepting the scan string but refusing the exploit, while the other due to a web application echoing only the initial scan string marker, causing scan string subsitution to fail. We believe that only these last two errors represent shortcomings of the scanner's approach.

Additionally, we retrieved the Google pagerank popularity of each sampled website, to see whether more famous websites are quicker in fixing their vulnerabilities. The results are shown in figure 5. Unexpectedly, popu-

larity and vulnerability rate do not seem to be related, at least for the pagerank values plotted. Although we have data for websites with pagerank above 7/10, the number of samples is too low to guarantee statistically meaningful results. We also scanned government and military websites for the same phenomenon: the rate for unfixed vulnerabilities is 34.86%, which is not a significantly lower figure than the average. This suggests that many developers not educated about XSS vulnerabilities, who introduced a trivial vulnerability which caused their web application to be detected by Gd0rk a year ago, did not learn about the threat in the following months, regardless of the popularity of the web application they maintain. Alternatively, these might be web applications that are not actively maintained.

Furthermore, we gathered statistics about how often parameters are reflected in a certain context for each page and how often these reflected instances are vulnerable. Table 6 summarises the results. These show that, although the vast majority of reflected content is found in HTML attribute values and text, a context-aware filter can be helpful in automatically building exploits for other context which occur with non-negligible frequency.

We believe that our database contains a wealth of information that can be extracted by further analysis. For example, by clustering URLs based on parameter name similarity and sorting the clusters by size, we found new popular web applications frameworks that were not included in the set of CVE vulnerabilities used to bootstrap the crawl. In other words, Gd0rk is potentially able to find novel, undisclosed vulnerabilities in web application frameworks, while existing vulnerability scanners employing Google Search were limited to the set of vulnerabilities expressed by the input dorks, which cannot uncover novel vulnerabilities.

A preliminary version of Gd0rk targeted to SQL Injection attacks was run as well; using similar network resources, it found 389,684 vulnerabilities. This proves that Google Search can be used for many class of vulnerabilities when combined with a specific vulnerability scanner.

7 Related Work

Related work falls roughly into two categories: black-box web application vulnerability scanners and white-box source code analyzers.

[22] describes Secubat, perhaps the most similar tool to Gd0rk: it is a vulnerability scanner that crawls the web for HTML forms, probing their target URLs for XSS vulnerabilities and SQL injections by appropriately filling form values and submitting the form. Instead of driving the crawl using google results, Secubat starts from a single URL provided by the user and follows links contained in webpages. It can thus be used to scan a single-web application for vulnerabilities. However, if the crawler is configured to follow external links, it is able to perform a large-scale scan much like Gd0rk. SQL Injection vulnerabilities are detected by searching for known error messages in the response and XSS vulnerabilities by injecting an attack string and looking for a match in the response. Many other web application scanners exist. [9, 7, 15, 8, 41] use Google to search for vulnerabilities from a list of known dorks, while most scanners do not use Google but rather crawl a specific website for vulnerabilities. The latter category is more targeted to web developers wishing to audit their own web applications. [32] uses a database of previously known vulnerabilties, while other scanners are able to perform black-box analysis to detect application logic faults and report novel vulnerabilities: closed-source solutions [1, 30] do not offer much insight into their detection techniques; [5] offers a comparison of their capabilities. Many open-source web application scanners have been developed as well [28, 43]. [33] attempts to compare their accuracy.

White-box analysis techniques have also been developed: static analysis attempts to infer the data flow and the transformations applied to the untrusted inputs before they are transmitted back to the user. It offers the attractive capability of exhaustively examining every program path before deployment; black-box techniques can only obtain partial code coverage. Earlier work in this field attempted to consolidate previous static analysis research on static languages, while addressing issues that arise in dynamic languages [20, 21, 42, 40], which are almost exclusively used to build web applications. Recent work has tackled the problem of statically analyzing sanitization functions as well [2, 39], instead of relying on whitelisted functions that are presumed to correctly filter out attacks. However, static analysis tools can only provide information to a security-aware educated developer, who is then expected to close the vulnerability. Unfortunately, our data suggests that many developers are not willing to maintain web applications and fix security vulnerabilities.

8 Conclusions

This paper presented Gd0rk, a tool that employs Google's URL search capabilities to perform a large-scale scan for XSS vulnerabilities. The results show that a scan using modest network capabilities and human involvement is possible: we have collected more than 200.000 XSS vulnerabilities, many of which are still active today.

References

[1] ACUNETIX. Acunetix Web Security Scanner. http://www.acunetix.com/company/index.htm.

[2] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *IEEE Security and Privacy Symposium* (2008).

Context	JS Event	Text	Script tag	Attribute Name	Attribute Value	JavaScript URL	HTML Comment	Title
Reflected	3.50%	18.51%	7.61%	0.02%	45.26%	0.86%	2.53%	4.00%
Vulnerable	3.90%	39.84%	14.47%	0.05%	54.08%	2.03%	5.43%	11.73%

Figure 6: Context Frequency

[3] BARTH, A. Enable the XSS auditor by default. http://codereview.chromium.org/6507010/.

[4] BATES, D., BARTH, A., AND JACKSON, C. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *Proc. of the 19th International World Wide Web Conference* (2010).

[5] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 332–345.

[6] CORPORATION, T. M. CVE List. http://cve.mitre.org/cve/cve.html.

[7] CULT OF THE DEAD COW. Goolag Scan. http://w3.cultdeadcow.com/cms/apps.html, 2008.

[8] D3HYDR8. D3hydr8 Google SQL scanner. http://r00tsecurity.org/db/code/txt.php?id=26, 2008.

[9] D3HYDR8. D3hydr8 Google XSS scanner. http://darkcode.ath.cx/scanners/XSSscan.py, 2009.

[10] DARK READING. Phishers Enlist Google Dorks. http://www.darkreading.com/security/application-security/211201291/index.html, 2008.

[11] GIORGIO MAONE. NoScript. http://noscript.net/.

[12] GLYPE.COM. Glype Proxy Script. http://www.glype.com.

[13] HACKERS FOR CHARITY. Google Hacking Database. http://www.hackersforcharity.org/ghdb/, 2011.

[14] HERMAN, D. Parser API. https://developer.mozilla.org/en/SpiderMonkey/Parser_API.

[15] HOBOJOEY. SQL Injection Scanner - sqli_find.pl. http://sqlinjectiontools.com/2010/12/sqlinjectionscanner/, 2010.

[16] INC., T. T. P. Tor Project: Anonymity Online. https://www.torproject.org.

[17] ISMAIL, O., ETOH, M., KADOBAYASHI, Y., AND YAMAGUCHI, S. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA04)* (2004).

[18] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, p. 610.

[19] JOHNS, M., ENGELMANN, B., AND POSEGGA, J. XSSDS: Server-side Detection of Cross-site Scripting Attacks. In *24th Annual Computer Security Applications Conference, Anaheim, CA, USA* (2008).

[20] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper).

[21] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security* (New York, NY, USA, 2006), PLAS '06, ACM, pp. 27–36.

[22] KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIC, N. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web* (2006), ACM, pp. 247–256.

[23] NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium* (2009).

[24] NAVA, E. V., AND LINDSAY, D. Our favorite xss filters/ids and how to attack them. Black Hat USA 2009.

[25] NAVA, E. V., AND LINDSAY, D. Universal xss via ie8's xss filters. Black Hat Europe 2010.

[26] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank Citation Ranking: Bringing Order to the Web.

[27] PROJECT, T. F. flex: The Fast Lexical Analyzer. http://flex.sourceforge.net/.

[28] RIANCHO, A. w3af. http://w3af.sourceforge.net/.

[29] ROSS, D. IE 8 XSS Filter Architecture/Implementation. http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[30] SECURITY, W. WhiteHat Security. https://www.whitehatsec.com/services/services.html.

[31] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 921–930.

[32] SULLO, C., AND LODGE, D. Nikto2. http://cirt.net/nikto2.

[33] TEODORO, N., AND SERRÃO, C. Automating Web Applications Security Assessments through Scanners. *Web Application Security*, 48.

[34] TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. *University of Illinois at Chicago, Tech. Rep* (2009).

[35] THE MITRE CORPORATION. 2010 CWE/SANS Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25/.

[36] THE OPEN WEB APPLICATION SECURITY PROJECT (OWASP). OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[37] VAN GUNDY, M., AND CHEN, H. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *16th Annual Network & Distributed System Security Symposium, San Diego, CA, USA* (2009).

[38] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS07)* (2007), Citeseer.

[39] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007), ACM, pp. 32–41.

[40] WASSERMANN, G., AND SU, Z. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering* (2008), ACM, pp. 171–180.

[41] WIKIMEDIA INC. Asprox Botnet. http://en.wikipedia.org/wiki/Asprox_botnet, 2008.

[42] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium* (2006), pp. 179–192.

[43] ZALEWSKI, M. skipfish. http://code.google.com/p/skipfish/.