deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation

Adam Doupé
UC Santa Barbara
adoupe@cs.ucsb.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Mariusz H. Jakubowski
Microsoft Research
mariuszj@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Web applications are constantly under attack. They are popular, typically accessible from anywhere on the Internet, and they can be abused as malware delivery systems.

Cross-site scripting flaws are one of the most common types of vulnerabilities that are leveraged to compromise a web application and its users. A large set of cross-site scripting vulnerabilities originates from the browser's confusion between data and code. That is, untrusted data input to the web application is sent to the clients' browser, where it is then interpreted as code and executed. While new applications can be designed with code and data separated from the start, legacy web applications do not have that luxury.

This paper presents a novel approach to securing legacy web applications by automatically and statically rewriting an application so that the code and data are clearly separated in its web pages. This transformation protects the application and its users from a large range of server-side cross-site scripting attacks. Moreover, the code and data separation can be efficiently enforced at run time via the Content Security Policy enforcement mechanism available in modern browsers.

We implemented our approach in a tool, called DEDACOTA, that operates on binary ASP.NET applications. We demonstrate on six real-world applications that our tool is able to automatically separate code and data, while keeping the application's semantics unchanged.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]

Keywords

Static Analysis; Cross-Site Scripting; XSS; Content Security Policy; CSP

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516708.

1. INTRODUCTION

Web applications are prevalent and critical in today's computing world, making them a popular attack target. Looking at types of vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [11], web application flaws are by far the leading class.

Modern web applications have evolved into complex programs. These programs are no longer limited to server-side code that runs on the web server. Instead, web applications include a significant amount of JavaScript code that is sent to and executed on the client. Such client-side components not only provide a rich and fast user interface, they also contain parts of the application logic and typically communicate with the server-side component through asynchronous JavaScript calls. As a result, client-side scripts are an integral component of modern web applications, and they are routinely generated by server-side code.

There are two kinds of cross-site scripting (XSS) vulnerabilities: server-side and client-side. The latter is essentially caused by bugs in the client-side code, while the former is caused by bugs in the server-side code. In this paper we focus on server-side XSS vulnerabilities (unless specified otherwise, we will use XSS to refer to server-side XSS). XSS vulnerabilities allow attackers to inject client-side scripting code (typically, JavaScript) into the output of web applications. The scripts are then executed by the browser as it renders the page, allowing malicious code to run in the context of the web application. Attackers can leverage XSS attacks to leak sensitive user information, impersonate the victim to perform unwanted actions in the context of the web application, or launch browser exploits.

There has been a significant amount of research effort on eliminating XSS vulnerabilities. The main line of research has focused on sanitizing untrusted input [3, 13, 18, 22, 25, 27, 34, 38, 40, 44, 46, 48–50]. Sanitization attempts to identify and "clean up" untrusted inputs that might contain JavaScript code. Performing correct sanitization is challenging, for a number of reasons. One reason is that it is difficult to guarantee coverage for all possible paths through the application [3, 48]. As part of this problem, it is necessary to find all program locations (sources) where untrusted input can enter the application, and then verify, along all program paths, the correctness of all sanitization functions that are used before the input is sent to the client (sinks). Furthermore, it is not always clear how to properly sanitize data, because a single input might appear in different contexts in the output of the application [40].

The root cause of XSS vulnerabilities is that *the current web application model violates the principle of code and data separation.* In the case of a web page, the data is the HTML content of the page and the code is the JavaScript code. Mixing JavaScript code and HTML data in the same channel (the HTTP response) makes it possible for an attacker to convince a user's browser to interpret maliciously crafted HTML data as JavaScript code. While sanitization tries to turn untrusted input, which could potentially contain code, into HTML data, we believe the fundamental solution to XSS is to separate the code and data in a web page—the way HTML and JavaScript should have been designed from the start. Once the code and data are separated, a web application can communicate this separation to the browser, and the browser can ensure no code is executed from the data channel. Such communication and enforcement is supported by the new W3C browser standard Content Security Policy (CSP) [42].

While new web applications can be designed with code and data separated from the start, it has been a daunting task to achieve code and data separation for legacy applications. The key challenge is to identify code or data in the output of a web application. Previous solutions have relied on either developers' manual annotations or dynamic analysis. For example, BEEP [20] requires developers to manually identify inline JavaScript code. BLUEPRINT [28] requires developers to manually identify the data by specifying which application statements could output untrusted input. XSS-GUARD dynamically identifies application-intended JavaScript code in a web page by comparing it with a shadow web page generated at run time [4]. The main problem preventing these solutions from being adopted is either the significant manual effort required from application developers or the significant runtime performance overhead. In fact, Weinberger et al. [47] showed how difficult it is to manually separate the code and data of a web application.

In this paper, we present DEDACOTA, the first system that can automatically and statically rewrite an existing web application to separate code and data in its web pages. Our novel idea is to use static analysis to determine all inline JavaScript code in the web pages of an application. Specifically, DEDACOTA performs static data-flow analysis of a given web application to approximate its HTML output. Then, it parses each page's HTML output to identify inline JavaScript code. Finally, it rewrites the web application to output the identified JavaScript code in a separate JavaScript file.

The problem of statically determining the set of (HTML) outputs of a web application is undecidable. However, as we observe in our evaluation, the problem is typically tractable for real-world web applications. These applications are written by benign developers and tend to have special properties that allow us to compute their outputs statically. For instance, the majority of the inline JavaScript code is static in the web applications we tested.

Dynamic inline JavaScript presents a second-order problem. Here, the JavaScript code itself (rather than the HTML page) is generated dynamically on the server and may depend on untrusted inputs. Again, the potential for XSS vulnerabilities exists. DEDACOTA provides a partial solution to this problem by producing alerts for all potentially dangerous instances of dynamic JavaScript generation in the application and by safely sanitizing a large subclass of these instances.

We implemented a prototype of DEDACOTA to analyze and rewrite ASP.NET [31] web applications. We then applied DEDACOTA to six open-source, real-world ASP.NET applications. We verified that all known XSS vulnerabilities are eliminated. We then performed extensive testing to ensure that the rewritten binaries still function correctly. We also tested DEDACOTA's performance and found that the page loading times between the original and rewritten application are indistinguishable.

The main contributions of this paper are the following:

- A novel approach for automatically separating the code and data of a web application using static analysis (Section 4).
- A prototype implementation of our approach, DEDACOTA, applied to ASP.NET applications (Section 5).
- An evaluation of DEDACOTA, showing that we are able to apply our analysis to six real-world, open-source, ASP.NET applications. We show that our implementation prevents the exploitation of know vulnerabilities and that the semantics of the application do not change (Section 6).

2. BACKGROUND

In this section, we provide the background necessary for understanding the design of DEDACOTA.

2.1 Cross-Site Scripting

Modern web applications consist of both server-side and client-side code. Upon receiving an HTTP request, the server-side code, which is typically written in a server-side language, such as PHP or ASP.NET, dynamically generates a web page as a response, based on the user input in the request or data in a backend database. The client-side code, which is usually written in JavaScript and is executed by the browser, can be either inline in the web page or external as a standalone JavaScript file.

Cross-site scripting (XSS) vulnerabilities allow an attacker to inject malicious scripts into web pages to execute in the client-side browser, as if they were generated by the trusted web site. If the vulnerability allows the attacker to store malicious JavaScript on the server (e.g., using the contents of a message posted on a newsgroup), the vulnerability is traditionally referred to as "stored" or "persistent XSS." When the malicious code is included in the request and involuntarily reflected to the user (copied into the response) by the server, the vulnerability is called "reflected XSS." Finally, if the bug is in the client-side code, the XSS vulnerability is referred to as "DOM-based XSS" [24]. We call the first two types of vulnerabilities "server-side XSS vulnerabilities" and the latter "client-side XSS vulnerabilities."

The root cause for server-side XSS is that the code (i.e., the client-side script) and the data (i.e., the HTML content) are mixed together in a web page. By crafting some malicious input that will be included into the returned web page by the server-side code, an attacker can trick the browser into confusing his data as JavaScript code.

2.2 Code and Data Separation

The separation of code and data can be traced back to the Harvard Architecture, which introduces separate storage and buses for code and data. Separating code and data is a basic security principle for avoiding code injection attacks [19]. Historically, whenever designs violate this principle, there exists a security hole. An example is the stack used in modern CPUs. The return addresses (code pointers) and function local variables (data) are co-located on the stack. Because the return addresses determine control transfers, they are essentially part of the code. Mixing them together with the data allows attackers to launch stack overflow attacks, where data written into a local variable spills into an adjacent return address. In the context of web applications, we face the same security challenge, this time caused by mixing code and data together in web pages. To fundamentally solve this problem, we must separate code and data in web pages created by web applications.

2.3 Content Security Policy

Content Security Policy (CSP) [42] is a mechanism for mitigating a broad class of content injection vulnerabilities in web applications. CSP is a declarative policy that allows a web application to inform the browser, via an HTTP header, about the sources from which the application expects to load resources such as JavaScript code. A web browser that implements support for CSP can enforce the security policy declared by the web application.

A newly developed web application can leverage CSP to avoid XSS by not using inline JavaScript and by specifying that only scripts from a set of trusted sites are allowed to execute on the client. Indeed, Google has required that all Chrome browser extensions implement CSP [1]. However, manually applying CSP to a legacy web application typically requires a non-trivial amount of work [47]. The reason is that the authors of the web application have to modify the server-side code to clearly identify which resources (e.g., which JavaScript programs) are used by a web page. Moreover, these scripts have to be separated from the web page.

CSP essentially provides a mechanism for web browsers to enforce the separation between code and data as specified by web applications. Our work solves the problem of automatically transforming legacy web applications so that the code and data in their web pages are separated. The transformed web applications can then directly leverage the browser's support for CSP to avoid a wide range of XSS vulnerabilities.

3. THREAT MODEL

Before discussing the design of DEDACOTA, we need to state our assumptions about the code that we are analyzing and the vulnerabilities we are addressing.

Our approach involves rewriting a web application. This web application is written by a benign developer—that is, the developer has not intentionally obfuscated the code as a malicious developer might. This assumption also means that the JavaScript and HTML are benign and not intentionally taking advantage of browser parsing quirks (as described in BLUEPRINT [28]).

DEDACOTA will only prevent *server-side* XSS vulnerabilities. We define server-side XSS vulnerabilities as XSS vulnerabilities where the *root cause* of the vulnerability is in server-side code. Specifically, this means XSS vulnerabilities where unsanitized input is used in an HTML page. We explicitly do not protect against client-side XSS vulnerabilities, also called DOM-based XSS. Client-side XSS vulnerabilities occur when untrusted input is interpreted as JavaScript by the client-side JavaScript code using methods such as `eval`, `document.write`, or `innerHTML`. The root cause of these vulnerabilities is in the *JavaScript code*.

In this paper, we focus solely on separating inline JavaScript code (that is, JavaScript in between `<script>` and `</script>`). While there are other vectors where JavaScript can be executed, such as JavaScript code in HTML attributes (event handlers such as `onclick`) and inline Cascading Style Sheet (CSS) styles [16], the techniques described here can be extended to approximate and rewrite the HTML attributes and inline CSS.

Unfortunately, code and data separation in an HTML page is not a panacea for XSS vulnerabilities. In modern web applications, the inline JavaScript code is sometimes dynamically generated by the server-side code. A common scenario is to use the dynamic JavaScript code to pass data from the server-side code to the client-side code. There may be XSS vulnerabilities, even if code and data are properly separated, if the data embedded in the JavaScript code is not properly sanitized. DEDACOTA provides a partial solution to the problem of dynamic JavaScript (see Section 4.5).

4. DESIGN

Our goal is to statically transform a given web application so that the new version preserves the application semantics but outputs web pages where all the inline JavaScript code is moved to external JavaScript files. These external files will be the *only* JavaScript that the browser will execute, based on a Content Security Policy.

There are three high-level steps to our approach. For each web page in the web application: (1) we statically determine a conservative approximation of the page's HTML output, (2) we extract all inline JavaScript from the approximated HTML output, and (3) we rewrite the application so that all inline JavaScript is moved to external files.

Hereinafter, we define a running example that we use to describe how DEDACOTA automatically transforms a web application, according to the three steps outlined previously.

4.1 Example

Listing 1 shows a simplified ASP.NET Web Form page. Note that everything not in between the `<%` and `%>` is output directly to the browser. Everything between matching `<%` and `%>` is C# code. A subtle but important point is that `<%=` is used to indicate that the C# code will output a string at that location in the HTML output.

In Listing 1, Line 2 sets the title of the page, and Line 3 sets the `Username` variable to the `name` parameter sent in the query string. The `Username` is output to the browser inside a JavaScript string on Line 7. This is an example of the C# server-side code passing information to the JavaScript client-side code, as the intent here is for the JavaScript `username` variable to have the same value as the C# `Username` variable.

Internally, ASP.NET compiles the ASP.NET Web Form page to C#, either when the application is deployed, or on-demand, as the page is accessed. The relevant compiled C# output of Listing 1 is shown in Listing 2. Here, the ASP.NET Web Form page has been transformed into an

```
1  <html>
2    <% Title = "Example";
3     Username = Request.Params["name"]; %>
4    <head><tile><%= Title %></title></head>
5    <body>
6      <script>
7        var username = "<%= Username %>";
8      </script>
9    </body>
10 </html>
```

Listing 1: Example of a simple ASP.NET Web Form page.

```
1  void Render(TextWriter w) {
2    w.Write("<html>\n  ");
3    this.Title = "Example";
4    this.Username = Request.Params["name"];
5    w.Write("\n  <head><tile>");
6    w.Write(this.Title);
7    w.Write("</title></head>\n  <body>\n
         <script>\n      var username = \"");
8    w.Write(this.Username);
9    w.Write("\";\n    </script>\n  </body>\n
         </html>");
10 }
```

Listing 2: The compiled C# output of Listing 1.

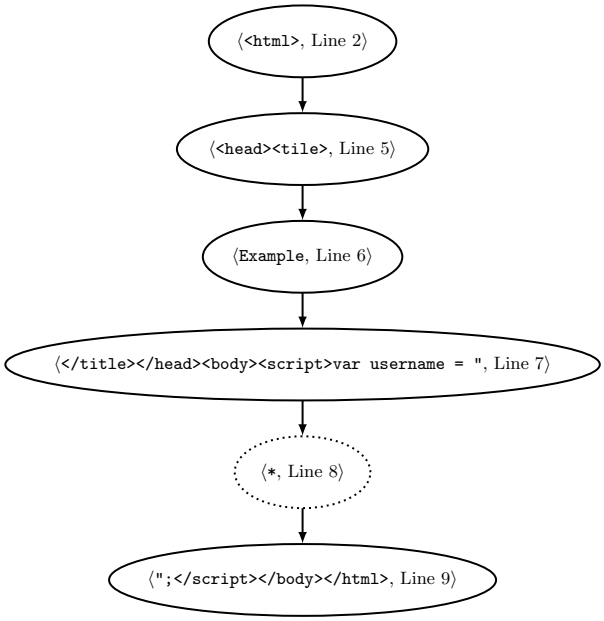

Figure 1: Approximation graph for the code in Listing 1 and Listing 2. The dotted node's content is not statically determinable.

equivalent C# program. The ASP.NET compiler creates a class (not shown) that represents the ASP.NET Web Form. A method of the class is given a `TextWriter` object as a parameter. Anything written to this object will be sent in the HTTP response. `TextWriter.Write` is a method call equivalent of writing to the console in a traditional command-line application.

From comparing Listing 1 to Listing 2, one can see that output not between `<%` and `%>` tags is written to the `TextWriter` object. The code between the `<%` and `%>` tags is inlined into the function (Lines 3 and 4), and the code that is between the `<%=` and `%>` tags is written to the `TextWriter` object (Lines 6 and 8). We also note that `TextWriter.Write` is one of a set of methods used to write content to the HTTP response. However, for simplicity, in the remainder of this paper, we will use `TextWriter.Write` to represent all possible ways of writing content to the HTTP response.

4.2 Approximating HTML Output

In the first phase of our approach, we approximate the HTML output of a web page. This is a two-step process. First, we need to determine, at every `TextWriter.Write` location, what is being written. Second, we need to determine the order of the `TextWriter.Write` function invocations.

We use a different static analysis technique to answer each of the two questions. To determine what is being written at a `TextWriter.Write`, we use the points-to analysis algorithm presented in [10] modified to work on .NET byte-code, instead of C. This points-to analysis algorithm is inclusion-based, demand-driven, context-sensitive, field-sensitive, and partially flow-sensitive. The points-to analysis algorithm computes the set of strings that alias with the parameter of `TextWriter.Write`. If all strings in the alias set are constant strings, the output at the `TextWriter.Write` will be defined as the conjunction of all possible constant strings. Otherwise, we say the output is statically undecidable. To determine the ordering of all `TextWriter.Write` method calls, we build a control-flow graph, using standard techniques, that only contains the `TextWriter.Write` method calls.

We encode the information produced by the two static analyses—the ordering of `TextWriter.Write` method calls and their possible output—into a graph that we call an *approximation graph.* Figure 1 shows the approximation graph for the code in Listing 1 and Listing 2. Each node in the graph contains the location of the `TextWriter.Write` that this node represents as well as the possible constant strings that could be output at this `TextWriter.Write` location. Content that cannot be determined statically is represented by a wild card `*` (the dotted node in Figure 1). The strings that may be output at the `TextWriter.Write` will be used to identify inline JavaScript, and the location of the `TextWriter.Write` will be used for rewriting the application.

In Figure 2 we show the approximation graph of a more complex page. The graph in Figure 2 contains a branch, where each node in the branch maps to the same `TextWriter.Write` method. This happens when the points-to analysis says that the `TextWriter.Write` method can output one of multiple strings. The other way there can be a branch in the approximation graph is when there is a branch in the control flow of the web application. The graph in Figure 2 also contains a loop that includes the nodes shown in bold. However, because we cannot statically determine the number of times a loop may execute, and we want our analysis to be conservative, we collapse all nodes of a loop (in the approximation graph) into a single node. This new node now has undecidable content (represented by a `*`). The new node also keeps track of all the `TextWriter.Write` methods that were part of the original loop.

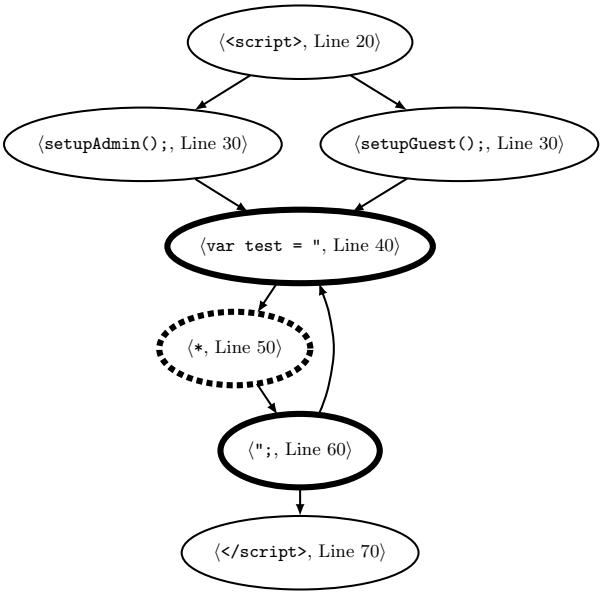

Figure 2: Approximation graph with branches and a loop. The loop will be collapsed into one node to create the final approximation graph.

After collapsing all loops in the graph, we derive a conservative approximation of the HTML output of a web page. The approximation graph is a directed acyclic graph (DAG), and any path from the root node to a leaf node will represent one possible output of the web page.

4.3 Extracting Inline JavaScript

In the second phase, our approach uses the approximation graph described previously to extract all possible inline JavaScript. The output of this phase is a set containing all possible inline JavaScript that may appear in the web page.

In an approximation graph, each unique path from the root node to a leaf node represents a potential output of the page. A naïve algorithm would enumerate all paths and, thus, all outputs, and parse each output string to identify inline JavaScript. However, even without loops, the number of unique paths even in a simple web page may quickly explode and become unmanageable (this is the path-explosion problem faced in static analysis).

To reduce the impact of the path explosion problem, we extract the inline JavaScript directly from the approximation graph. We first search for the opening and closing tags of HTML elements in the graph. We ignore tags that appear in comments. Then, for each pair of JavaScript tags (i.e., `<script>` and `</script>`), we process all the unique paths between the opening and closing tags. For each path, we obtain an inline JavaScript that the program might output.

While our current prototype is relatively simplistic in parsing the starting and ending JavaScript files, it could be possible to use the parsing engine from a real browser. However, this is not as straight-forward as it seems, as our input is a graph of all potential HTML output, not a single document. We leave this approach to future work.

All identified inline JavaScript pieces are then passed to the last phase of our approach, which decides how to rewrite the application.

4.4 Application Rewriting

The goal of the third phase is to rewrite the application so that all identified inline JavaScript will be removed from the HTML content and saved in external JavaScript files. In the HTML code, an inline JavaScript is replaced with a reference to the external JavaScript file as follows:

```
<script src="External.js"></script>
```

It is not uncommon that multiple possible inline JavaScript snippets exist between an opening and closing JavaScript tag because there may be branches between the tags in the approximation graph. To know which exact inline JavaScript is created, we need to track the execution of the server-side code.

The inline JavaScript identified in the previous phase falls into two categories: static and dynamic (i.e., contains undecidable content). Because we cannot statically decide the content of a dynamic inline JavaScript, we must track the execution of the server-side code to create its external JavaScript file(s) at runtime. Therefore, we can avoid tracking the execution of the server-side code *only* for the case in which there is a *single, static* inline JavaScript code.

For a pair of opening and closing script tags that require tracking the execution of the server-side code, we rewrite the application as follows. At the `TextWriter.Write` that may output the opening script tag, we first check if the output string contains the tag. We need to perform this check because a `TextWriter.Write` site may be used to output either inline JavaScript code or other HTML. If we find the opening script tag in the output, we use a session flag to indicate that an inline JavaScript rewriting has started. We write out everything before the start of the opening script tag. We remove the opening script tag itself. The remaining content is stored into a session buffer. Note that both session flag and buffer are unique to each opening script tag. Then, for all subsequent `TextWriter.Write` method calls that are part of the inline JavaScript we are rewriting, except for the last (that writes the closing tag), we append their output to the session buffer if the session flag is on. For the last `TextWriter.Write` method call (i.e., the one that writes the closing script tag), any string content that occurs before the closing script tag is appended to the session buffer. Any content after the closing script tag is just written to the output. At this point, the session buffer contains the entire inline JavaScript code. We save this code to an external file and add a `TextWriter.Write` method call that outputs the reference to this JavaScript file.

To support JavaScript caching on the client side, the name of the JavaScript file is derived from its content, using a cryptographic hash of the JavaScript content. An unintended benefit of this approach is that inline JavaScript that is included on multiple pages will be cached by the browser, improving application performance by reducing the size of the page and saving server requests.

Listing 3 shows the result of applying this rewriting process to the inline JavaScript code in Listing 2. The changes shown are only those made to Lines 7–9 in Listing 2.

4.5 Dynamic Inline JavaScript

At this point in our analysis, we have successfully separated the JavaScript code from the HTML data in the web application. If the web application's JavaScript is static, and by static we mean statically decidable, then the application is now immune to XSS vulnerabilities. However, if

```
1  w.Write("</title></head>\n  <body>\n   ");
2
3  Session["7"] = "\n    var username = \"";
4  Session["7"] += this.Username;
5  Session["7"] += "\";\n    ";
6
7  var hashName = Hash(Session["7"]) + ".js";
8  WriteToFile(hashName, Session["7"]);
9
10 w.Write("<script src=\"" + hashName + "
       \"></script>");
11
12 w.Write("\n  </body>\n</html>");
```

Listing 3: The result of the rewriting algorithm applied to Listing 2. Specifically, here we show the transformation of Lines 7–9 in Listing 2.

the web application dynamically generates JavaScript with undecidable content, and that content is not properly sanitized inside the JavaScript code, an attacker can exploit this bug to inject a malicious script. The approach discussed so far does not mitigate this attack, because it simply moves the vulnerable JavaScript to an external file.

To understand how dynamic JavaScript can result in a vulnerability, consider our example application in Listing 2. There is an XSS vulnerability on Line 8 because the `Username` variable is derived from the `name` parameter and output directly to the user, without sanitization. An attacker could exploit this vulnerability by setting the `name` parameter to `";alert('xss')//`. This would cause the resulting inline JavaScript to be the following, thus executing the attacker's JavaScript code:

```
<script>
  var username = "";alert('xss')//";
</script>
```

Therefore, the code section of the application is dynamically generated with untrusted input and even with the code and data separated, there is still an XSS vulnerability.

We attempt to mitigate this problem, and therefore improve the security of the application, in two ways. First, we identify cases in which we can safely rewrite the application. Second, we notify the developer when we make an inline to external transformation that is potentially unsafe.

For the first case, when the undetermined output is produced in certain JavaScript contexts, we can include it in a safe fashion via sanitization. Specifically, during static analysis we pass the dynamic inline JavaScript to a JavaScript parser. Then, we query the parser to determine the contexts in which the undetermined output (i.e., the `*` parts) is used. Here, for context we are referring specifically to the HTML parsing contexts described by Samuel et al. [38]. Possible contexts are JavaScript string, JavaScript numeric, JavaScript regular expression, JavaScript variable, etc. If an undetermined output is in a string context, we sanitize them in a way similar to how BLUEPRINT [28] handles string literals in JavaScript.

Like BLUEPRINT, on the server side we encode the string value and store the encoded data in JavaScript by embedding a call to a decoding function. Then when the JavaScript is executed on the client side, the decoding function will decode the encoded data and return the string. Unlike BLUEPRINT, we do not require any developer annotations because our static analysis can automatically identify which JavaScript context an undetermined output is in.

4.6 Generality

While the description of our approach so far was specific to ASP.NET Web Forms, the high-level idea of automatically separating code and data in a legacy web application can be generalized to any other web application frameworks or templating languages. There are still challenges that remain to apply our approach to another language, or even another template in the same language. The two main steps of our approach that must be changed to accommodate a different language or templating language are: (1) understand how the output is created by the web application and (2) understand how to rewrite the web application. Only the first step affects the analysis capability (as the rewriting process is fairly straightforward).

To automatically separate the code and data of a different language or templating language, one must understand how the language or template generates its output. After that, one would need to implement a static analysis that can create an approximation graph. For instance, in the default Ruby on Rails template, ERB, variables are passed to the template either via a hash table or class instance variables [37]. Therefore, one could approximate the output of an ERB template by statically tracking the variables added to the hash table and class instance variables (using points-to analysis). Once an approximation graph is created, detecting inline JavaScript can be performed in the manner previously described.

The main factor to affect the success of applying our approach to another web application framework or templating language is the precision of the static analysis, or in other words, how precise and detailed the approximation graph would be. The more dynamicism in the language or framework, such as run-time code execution and dynamic method invocation, the more difficult the analysis will be. Simply, the more of the control-flow graph that we are able to determine statically, the better our analysis will be. As an example the default templating language in Django only allows a subset of computation: iterating over a collection instead of arbitrary loops [12]. This restriction could make the analysis easier and therefore the approximation graph more precise.

5. IMPLEMENTATION

We implemented the automated code and data separation approach described in Section 4 in a prototype called DEDACOTA. This prototype targets ASP.NET Web Forms applications. ASP.NET is a widely used technology; of the Quantcase top million websites on the Internet, 21.24% use ASP.NET [8].

DEDACOTA targets *binary* .NET applications. More precisely, it takes as input ASP.NET Web Forms binary web applications, performs the three steps of our approach, and outputs an ASP.NET binary that has all inline JavaScript code converted into external JavaScript files. We operate at the binary level because we must be able to analyze the ASP.NET system libraries, which are only available in binary form.

We leverage the open-source Common Compiler Infrastructure (CCI) [32] for reading and analyzing the .NET Common Language Runtime byte-code. CCI also has mod-

ules to extract basic blocks and to transform the code into single static assignment (SSA) form. We also use CCI to rewrite the .NET binaries.

For the static analysis engine, we leverage the points-to analysis engine of KOP (also known as MAS) [10]. KOP was originally written for the C programming language. Therefore, we wrote (using CCI) a frontend that processes .NET binaries and outputs the appropriate KOP points-to rules. Then, after parsing these rules, the static analysis engine can provide either alias analysis or points-to analysis. The KOP points-to analysis is demand-driven, context-sensitive, field-sensitive, and, because of the CCI single static assignment, partially flow-sensitive.

An important point, in terms of scalability, is the demand-driven ability of the static analysis engine. Specifically, we will only explore those parts of the program graph that are relevant to our analysis, in contrast to traditional data-flow techniques which track data dependencies across the entire program. The demand-driven nature of the static analysis engine offers another scalability improvement, which is parallelism. Each analysis query is independent and, therefore, can be run in parallel.

We also extend the KOP points-to analysis system to model string concatenation. We do this by including special edges in the program graph that indicate that a variable is the result of the concatenation of two other variables. When computing the alias set of a variable, we first do so in the original way (ignoring any concatenation edges). Then, for each variable in the alias set that has concatenation edges, we compute the alias set for each of the two variables involved in the concatenation operation. We concatenate strings in the two alias sets and add them to the original alias set. The undecidable variables are tracked, so that their concatenated result contains a wildcard. This process is recursive, and handles arbitrary levels of concatenation.

ASP.NET uses the idea of reusable components, called `Controls`. The idea is that a developer can write a control once and then include it in other pages, and even other controls. This relationship of including one control inside another creates a parent-child relationship between the controls (the parent being the control that contains the child control).

In an ASP.NET Web Form page, child controls are first added to the parent's `ChildControls` collection, which is similar to an array. Then, during rendering, a parent renders its child controls either by iterating over the `ChildControls` or by referencing a child control based on its index in the `ChildControls`. Because the KOP points-to analysis does not model the array relation, we cannot precisely decide which child Control is being selected during rendering. To handle this problem, we need to track the parent-child relationships directly.

These parent-child relationships form a tree. Figure 3 shows the parent-child relationship of some of the user controls of `default.aspx` in the application BlogEngine.NET (one of the programs used in our evaluation). When building the control graph, we must statically recreate this tree.

To create this relationship statically, we take an approach similar to approximating the HTML output. The entry function for an ASP.NET page is `FrameworkInitialize`, which is similar to the `main` function for a C program. Starting from this method, we create a control-flow graph of all the calls to `AddParsedSubObject`, which is the function that adds a child control to a parent. This gives us the order of the `AddParsedSubObject` calls. At each of the calls, we use the points-to analysis to find which control is the parent and which is the child. This information, along with the order of the calls to `AddParsedSubObject`, allows us to recreate the parent-child control tree.

6. EVALUATION

There are three properties that we must look at to evaluate the effectiveness of DEDACOTA. First, do we prevent XSS vulnerabilities in the data section of the application by applying code and data separation? Second, do we correctly separate the code and data of the application—that is, does the rewriting preserve the application's semantics? Third, what is the impact on the application's performance? To evaluate the security of our approach, we look at ASP.NET applications with known vulnerabilities. To evaluate the correctness of our rewriting procedure, we apply our approach to applications that have developer-created integration tests. Then, we carried out performance measurements to answer the third question. Finally, we discuss the relation between separating code and data in the output and sanitizing the input.

6.1 Applications

We wish to evaluate DEDACOTA on ASP.NET web applications that are real-world, are open-source, and contain known vulnerabilities. Real-world applications are important for showing that our approach works on real-world code, open-source is important for other researchers to replicate our results, and known-vulnerable is important because we aim to automatically prevent these known vulnerabilities.

Unfortunately, there is no standard (or semi-standard) ASP.NET web application benchmark that meets all three requirements. Furthermore, finding these application proved to be a challenge. Compared to other languages such as PHP, there are fewer open-source ASP.NET applications (as most ASP.NET applications tend to be proprietary). Therefore, here we present a benchmark of six real-world, open-source, ASP.NET applications, four of which are known-vulnerable, one of which is intentionally vulnerable for education, and one of which has a large developer-created test suite.

Table 1 contains, for each application, the version of the application used in our evaluation, the CVE number of the vulnerability reported for the application, the number of ASP.NET Web Form pages, and the number of developer-written ASP.NET `Controls`. To provide an idea of the size of the applications, we also show the number of lines of code (LOC) of the ASP.NET controls (Web Forms and Controls) and C# code.

The web applications BugTracker.NET [7], BlogEngine.NET [5], BlogSA.NET [6], and ScrewTurn Wiki [41] all contain an XSS vulnerability as defined in the associated CVE.

WebGoat.NET [17] is an open-source ASP.NET application that is intentionally vulnerable. The purpose is to provide a safe platform for interested parties to learn about web security. Among the vulnerabilities present in the application are two XSS vulnerabilities.

ChronoZoom Beta 3 [9], is an open-source HTML5 "interactive timeline for all of history." Parts are written in

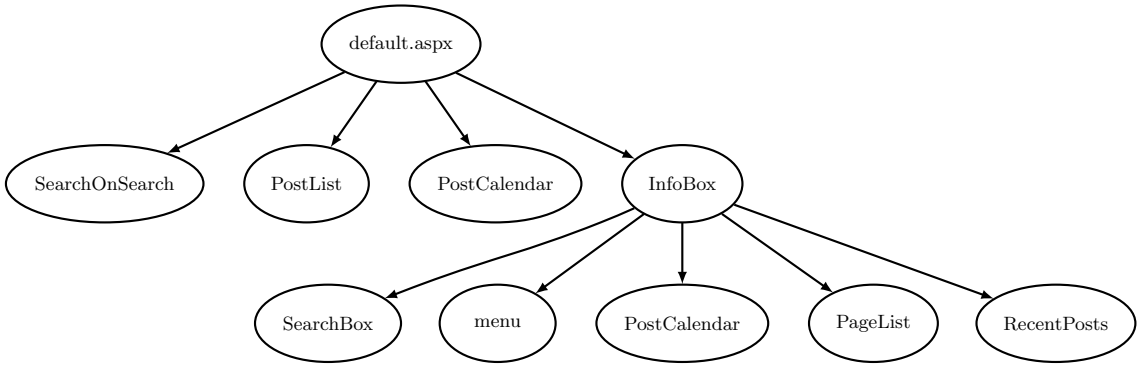

Figure 3: Control parent-child relationship between some of the controls in `default.aspx` from the application BlogEngine.NET. The siblings are ordered from left to right in first-added to last-added order.

Application	Version	Known Vulnerability	# Web Forms	# Controls	ASP.NET LOC	C# LOC	Total LOC
BugTracker.NET	3.4.4	CVE-2010-3266	115	0	27,257	8,417	35,674
BlogEngine.NET	1.3	CVE-2008-6476	19	11	2,525	26,987	29,512
BlogSA.NET	1.0 Beta 3	CVE-2009-0814	29	26	2,632	4,362	6,994
ScrewTurn Wiki	2.0.29	CVE-2008-3483	30	4	2,951	9,204	12,155
WebGoat.NET	e9603b9d5f	2 Intentional	67	0	1,644	10,349	11,993
ChronoZoom	Beta 3	N/A	15	0	3,125	18,136	21,261

Table 1: ASP.NET Web Form applications that we ran DEDACOTA on to test its applicability to real-world web applications.

ASP.NET Web Forms, but the main application is a JavaScript-heavy HTML page. We use ChronoZoom because, unlike the other applications, it has an extensive test suite that exercises the JavaScript portion of the application. To evaluate the correctness of our rewriting, we converted the main HTML page of ChronoZoom, which contained inline JavaScript, into an ASP.NET Web Form page, along with nine other HTML pages that were used by the test suite.

These six real-world web applications encompass the spectrum of web application functionality that we expect to encounter. These applications constitute a total of 100,000 lines of code, written by different developers, each with a different coding style. Some had inline JavaScript in the ASP.NET page, some created inline JavaScript in C# directly, while others created inline JavaScript in C# using string concatenation. Furthermore, while analyzing each application we also analyzed the entire .NET framework (which includes ASP.NET); all 256 MB of binary code. As our analysis handles ASP.NET, we are confident that our approach can be applied to the majority of ASP.NET applications.

6.2 Security

We ran DEDACOTA on each of our test applications. Table 2 shows the total number of inline JS scripts per application and a breakdown of the number of static inline JS scripts, the number of safe dynamic inline JS scripts, and the number of unsafe dynamic inline JS scripts. There were four dynamic inline JS scripts created by the ASP.NET framework, and these are represented in Table 2 in parentheses. We chose to exclude these four from the total dynamic inline JS scripts because they are not under the developer's control, and, furthermore, they can and should be addressed by changes to the ASP.NET library. Furthermore, it is important to note that our tool found these dynamic inline JS scripts within the ASP.NET framework automatically.

From our results it is clear that modern web applications frequently use inline JS scripts. The applications used a range of five to 46 total inline JS scripts. Of these total inline JS scripts 22% to 100% of the inline JS scripts were static.

DEDACOTA was able to safely transform, using the technique outlined in Section 4.5, 50% to 70% of the dynamic inline JS scripts. This result means that our mitigation technique worked in the majority of the cases, with only zero to four actual unsafe dynamic inline JS scripts per application.

We looked for false negatives (inline JavaScript that we might have missed) in two ways. We manually browsed to every ASP.NET Web Form in the application and looked for inline JavaScript. We also searched for inline JavaScript in the original source code of the application to reveal possible scripts the previous browsing might have missed. We did not find any false negatives in the applications.

To evaluate the security improvements for those applications that had known vulnerabilities, we manually crafted inputs to exploit these know bugs. After verifying that the exploits worked on the original version of the application, we launched them against the rewritten versions (with the Content Security Policy header activated, and with a browser supporting CSP). As expected, the Content Security Policy in the browser, along with our rewritten applications, successfully blocked all exploits.

6.3 Functional Correctness

To evaluate the correctness of our approach, and to verify that we maintained the semantics of the original application, we used two approaches. First, we manually browsed web pages generated by each rewritten application and interacted with the web site similar to a normal user. During this process, we looked for JavaScript errors, unexpected behaviors, or CSP violations. We did not find any problems or deviations. Second, and more systematically, we leveraged

Application	Total JS	Static	Safe Dynamic	Unsafe Dynamic
BugTracker.NET	46	41	3	2 (4)
BlogEngine.NET	18	4	10	4 (4)
BlogSA.NET	12	10	1	1 (4)
ScrewTurn Wiki	35	27	4	4 (4)
WebGoat.NET	6	6	0	0 (4)
ChronoZoom	5	5	0	0 (4)

Table 2: Results of running deDacota against the ASP.NET Web Form applications. Safe Dynamic is the number of dynamic inline JS scripts that we could safely transform, and Unsafe Dynamic is the number of dynamic inline JS scripts that we could not safely transform.

Application	Page Size	Loading Time
ChronoZoom (original)	50,827	0.65
ChronoZoom (transformed)	20,784	0.63
BlogEngine.NET (original)	18,518	0.15
BlogEngine.NET (transformed)	19,269	0.16

Table 3: Performance measurements for two of the tested applications, ChronoZoom. Page Size is the size (in bytes) of the main HTML page rendered by the browser, and Loading Time is the time (in seconds) that the browser took to load and display the page.

the developer-written testing suite in ChronoZoom. Before we applied our rewriting, the original application passed 160 tests. After rewriting, all 160 tests executed without errors.

6.4 Performance

To assess the impact of deDacota on application performance, we ran browser-based tests on original and transformed versions of two of the tested applications. Our performance metric was page-loading time in Internet Explorer 9.0, mainly to determine the impact of moving inline JavaScript into separate files. The web server was a 3 GB Hyper-V virtual machine running Microsoft IIS 7.0 under Windows Server 2008 R2, while the client was a similar VM running Windows 7. The physical server was an 8 GB, 3.16 GHz dual-core machine running Windows Server 2008 R2.

Table 3 shows test results for two web applications, summarizing performance data from page-loading tests on the client. The table columns list the average sizes of the main HTML pages retrieved by the browser by accessing the main application URLs, along with the average time used by the browser to retrieve and render the pages in their entirety. All the numbers were averaged over 20 requests.

As Table 3 indicates, deDacota's transformations incurred no appreciable difference in page-loading times. Because the original ChronoZoom page contained a significant amount of script code, the transformed page is less than half of the original size. On the other hand, the BlogEngine.NET page is slightly larger because of its small amount of script code, which was replaced by longer links to script files. The page-loading times mirror the page sizes, also indicating that server-side processing incurred no discernible performance impact.

6.5 Discussion

The results of our rewriting shed light on the nature of inline JavaScript in web applications. Of the four applications that have dynamic JavaScript, 12.2% to 77.8% of the total inline JavaScript in the application is dynamic. This is important, because one of BEEP's XSS prevention policies is a whitelist containing the SHA1 hash of allowed JavaScript [20]. Unfortunately, in the modern web JavaScript is not static and frequently includes dynamic elements, necessitating new approaches that can handle dynamic JavaScript.

The other security policy presented in BEEP is DOM sandboxing. This approach requires the developer to manually annotate the sinks so that they can be neutralized. Blueprint [28] works similarly, requiring the developer to annotate the outputs of untrusted data. Both approaches require the developer to manually annotate the sinks in the application in order to specify the trusted JavaScript. To understand the developer effort required to manually annotate the sinks in the application, we counted the sinks (i.e., `TextWriter.Write` call sites) inside the 29 Web Forms of BlogSA.NET and there were 407. In order to implement either BEEP or Blueprint a developer must manually analyze all sinks in the application and annotate any that could create untrusted output.

Unlike BEEP and Blueprint, deDacota is completely automatic and does not require any developer annotations. deDacota cannot prevent XSS vulnerabilities in dynamic inline JavaScript completely. If a developer wishes to prevent all XSS vulnerabilities after applying deDacota, they would only need to examine the sinks that occur *within* the unsafe dynamic inline JavaScript. In BlogSA.NET, there are three sinks within the single unsafe dynamic JavaScript. One could further reduce the number of sinks by using taint analysis to check if untrusted input can reach a sink in the dynamic JavaScript.

7. LIMITATIONS

The goal of deDacota is to automatically separate the JavaScript code from the HTML data in the web pages of a web application using static analysis. We have shown that deDacota is effective with real-world web applications. In this section, we discuss its limitations in general.

The programming language of .NET has the following dynamic language features: dynamic assembly loading, dynamic compilation, dynamic run-time method calling (via reflection), and threading. The use of these features may compromise the soundness of any static analysis including ours in deDacota. However, these language features are rarely used in ASP.NET web applications in practice. For instance, those applications we tested did not use any of these features. Furthermore, deDacota is affected only if the use of these features determines the HTML output of an application.

On one hand, we handle loops conservatively by approximating that a loop can produce anything. On the other hand, we treat the output of a loop as a `*` in the approximation graph and assume it does not affect the structure of the approximation graph in a way that impacts our analysis. For instance, we assume the output of a loop does not contain the opening or closing script tag. Our analysis will be incorrect if this assumption is violated. While we found that this assumption holds for all the web applications we tested, it is possible that this assumption will not hold for other programs, thus requiring a different approach to handling loops.

We do not offer any formal proof of the correctness of deDacota. While we believe that our approach is correct in

absence of the dynamic language features, we leave a formal proof of this to future work.

DEDACOTA currently supports the analysis of string concatenations. The support for more complex string operations such as regular expressions is left for future work. A potential approach is to leverage an automata-based string analysis engine [50].

Our approach to sanitizing dynamic JavaScript code may not preserve an application's semantics when the dynamic content being sanitized as a string is meant to be used in multiple JavaScript contexts.

When deploying DEDACOTA in practice, we recommend two practices to mitigate its limitations. First, all tests for the original web application should be performed on the rewritten binary to detect any disruptions to the application's semantics. Second, CSP's "Report Only" mode should be used during the testing and initial deployment. Under this mode, the browser will report violations back to the web server when unspecified JavaScript code is loaded. This helps detect inline JavaScript code that is missed by DEDACOTA.

Finally, our prototype does not handle JavaScript code in HTML attributes. We do not believe that there is any fundamental limitation that makes discovering JavaScript attributes more difficult than inline JavaScript. The only minor difficulty here is in the rewriting. In order to separate a JavaScript attribute into an external JavaScript, one must be able to uniquely identify the DOM element that the JavaScript attribute affects. To do this, it would require generating a unique identifier for the HTML element associated with the JavaScript attribute.

8. RELATED WORK

A broad variety of approaches have been proposed to address different types of XSS, though no standard taxonomy exists to classify these attacks and defenses. In general, XSS defenses employ schemes for input sanitization or restrictions on script generation and execution. Differences among various techniques involve client- or server-side implementation and static or dynamic operation. We group and review XSS defenses in this context.

8.1 Server-Side Methods

While CSP itself is enforced by browsers [42], our approach for leveraging CSP is a static, server-side XSS defense. There has been much previous research in server-side XSS defenses [3, 4, 13, 14, 22, 28, 34, 35, 38, 40, 43]. Server-based techniques aim for dynamically generated pages free of XSS vulnerabilities. This may involve validation or injection of appropriate sanitizers for user input, analysis of scripts to find XSS vulnerabilities, or automatic generation of XSS-free scripts.

Server-side sanitizer defenses either check existing sanitization for correctness or generate input encodings automatically to match usage context. For example, Saner [3] uses static analysis to track unsafe inputs from entry to usage, followed by dynamic analysis to test input cases for proper sanitization along these paths. SCRIPTGARD [40] is a complementary approach that assumes a set of "correct" sanitizers and inserts them to match the browser's parsing context. BEK [18] focuses on creating sanitization functions automatically analyzable for preciseness and correctness. Sanitization remains the main industry-standard defense against XSS and related vulnerabilities.

A number of server-side defenses restrict scripts included in server-generated pages. For example, XSS-GUARD [4] determines valid scripts dynamically and disallows unexpected scripts. The authors report performance overheads of up to 46% because of the dynamic evaluation of HTML and JavaScript. Templating approaches [13, 36, 38] generate correct-by-construction scripts that incorporate correct sanitization based on context. In addition, schemes based on code isolation [1,2,26] mitigate XSS by limiting DOM access for particular scripts, depending on their context.

Certain XSS defenses [21, 22, 27, 29, 34, 35, 39, 44, 49] use data-flow analysis or taint tracking to identify unsanitized user input included in a generated web page. These approaches typically rely on sanitization, encoding, and other means of separating unsafe inputs from the script code. Some schemes prevent XSS bugs dynamically, while others focus on static detection and elimination.

Other approaches [14, 28, 33] combine server-side processing with various client-side components, such as confinement of untrusted inputs and markup randomization. Such schemes may parse documents on the server and prevent any modifications of the resulting parse trees on the client. In addition, randomization of XHTML tags can render foreign script code meaningless, defeating many code-injection attacks.

8.2 Client-Side Methods

Client-side XSS defenses [20,23,30,42,45,47] mitigate XSS while receiving or rendering untrusted web content. Some of these schemes rely on browser modifications or plug-ins, often reducing their practical applicability. Others use custom JavaScript libraries or additional client-side monitoring software. CSP itself [42] is a browser-based approach, but its incorporation into WWW standards should facilitate wide acceptance and support by all popular browsers.

Some client-side XSS defenses focus on detecting and preventing leakage of sensitive data. For example, Noxes [23] operates as a personal-firewall plug-in that extracts all static links from incoming web pages, prompting the user about disclosure of information via dynamically generated links. Vogt et al. [45] also aim to address this problem, but use taint-tracking analysis within a browser to check for sensitive data released via XSS attacks. In contrast, DEDACOTA simply prevents any XSS exploits that could enable such leakage.

Client-side HTML security policies mitigate XSS via content restrictions, such as disallowing unsafe features or executing only "known good" scripts. Using a browser's HTML parser, BEEP [20] constructs whitelists of scripts, much like XSS-GUARD's server-side approach [4]. BEEP assumes no dynamic scripts whose hashes cannot be pre-computed, limiting its practicality with modern web applications; moreover, it has been shown that even whitelisted scripts may be vulnerable to attacks [2]. Another custom content security policy is BLUEPRINT's page descriptions, which are interpreted and rendered safely by a custom JavaScript library [28]. Script policies enforced at runtime [15, 30] are also useful for mitigating XSS exploits.

In general, standardized HTML security policies [42, 47] offer promise as a means of escaping the recent proliferation of complex, often ad hoc XSS defenses. CSP simplifies the

problem by enforcing fairly strong restrictions, such as disabling `eval()` and other dangerous APIs, prohibiting inline JavaScript, and allowing only local script resources to be loaded. While new web applications can be designed with CSP in mind, legacy code may require significant rewriting. deDacota works on both old and new applications, facilitating adoption of CSP by developers, primarily by automating the separation process.

9. CONCLUSION

Cross-site scripting vulnerabilities are pervasive in web applications. Malicious users frequently exploit these vulnerabilities to infect users with drive-by downloads or to steal personal information.

While there is currently no silver bullet to preventing every possible XSS attack vector, we believe that adhering to the fundamental security principle of code and data separation is a promising approach to combating XSS vulnerabilities. deDacota is a novel approach that gets us closer to this goal, by using static analysis to automatically separate the code and data of a web application. While not a final solution, deDacota and other tools that automate making web applications secure by construction are the next step in the fight against XSS and other kinds of vulnerabilities.

10. ACKNOWLEDGMENTS

The authors extend their thanks to David Molnar, Alex Moshchuk, Helen Wang, and Chris Hawblitzel for their helpful discussions, Herman Venter for all his help and support with CCI, and David Brumley for his insightful suggestions which helped to focus the paper. This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the Army Research Office (ARO) under Grant W911NF0910553, the National Science Foundation (NSF) under Grants CNS-0845559 and CNS-0905537, and Secure Business Austria.

11. REFERENCES

[1] Akhawe, D., Saxena, P., and Song, D. Privilege Separation in HTML5 Applications. In *Proceedings of the USENIX Security Symposium (USENIX)* (2012).

[2] Athanasopoulos, E., Pappas, V., and Markatos, E. P. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)* (2009).

[3] 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 *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, 2008).

[4] Bisht, P., and Venkatakrishnan, V. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (Paris, France, 2008).

[5] blogengine.net - an innovative open source blogging platform. `http://www.dotnetblogengine.net`, 2013.

[6] BlogSA.NET. `http://www.blogsa.net/`, 2013.

[7] BugTracker.NET - Free Bug Tracking. `http://ifdefined.com/bugtrackernet.html`, 2013.

[8] Top in Frameworks - Week beginning Jun 24th 2013. `http://trends.builtwith.com/framework`, 2013.

[9] Chronozoom - A Brief History of the World. `http://chronozoom.cloudapp.net/firstgeneration.aspx`, 2013.

[10] Cui, W., Peinado, M., Xu, Z., and Chan, E. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of the USENIX Security Symposium (USENIX)* (Bellevue, WA, 2012).

[11] CVE Details. Vulnerabilities by Type. `http://www.cvedetails.com/vulnerabilities-by-types.php`, 2013.

[12] Django. `http://djangoproject.com`, 2013.

[13] Google. Google AutoEscape for CTemplate. `http://code.google.com/p/ctemplate/`.

[14] Gundy, M. V., and Chen, H. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Network and Distributed System Security Symposium (NDSS)* (2009).

[15] Hallaraker, O., and Vigna, G. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)* (Shanghai, China, 2005).

[16] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[17] Hoff, J. WebGoat.NET. `https://github.com/jerryhoff/WebGoat.NET`, 2013.

[18] Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M. Fast and Precise Sanitizer Analysis with Bek. In *Proceedings of the USENIX Security Symposium (USENIX)* (2011).

[19] Howard, M., and LeBlanc, D. *Writing Secure Code*, second ed. Microsoft Press, 2003.

[20] Jim, T., Swamy, N., and Hicks, M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International World Wide Web Conference (WWW)* (2007).

[21] Johns, M., and Beyerlein, C. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (2007).

[22] Jovanovic, N., Kruegel, C., and Kirda, E. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2006).

[23] Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (2006).

[24] Klein, A. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, 2005.

[25] Livshits, B., and Chong, S. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)* (2013).

[26] Livshits, B., and Erlingsson, U. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2007).

[27] Livshits, V. B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium (USENIX)* (2005).

[28] Louw, M. T., and Venkatakrishnan, V. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).

[29] Martin, M., and Lam, M. S. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the USENIX Security Symposium (USENIX)* (2008).

[30] Meyerovich, L., and Livshits, B. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[31] Microsoft. ASP.NET. `http://www.asp.net/`.

[32] Microsoft Research. Common Compiler Infrastructure. `http://research.microsoft.com/en-us/projects/cci/`, 2013.

[33] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-Site Scripting Defense. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)* (2008).

[34] Nguyen-tuong, A., Guarnieri, S., Greene, D., and Evans, D. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the IFIP International Information Security Conference* (2005).

[35] Pietraszek, T., and Berghe, C. V. Defending against Injection Attacks through Context-Sensitive String Evaluations. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)* (2005).

[36] Robertson, W., and Vigna, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium (USENIX)* (Montreal, Quebec CA, 2009).

[37] Ruby on Rails. `http://rubyonrails.org/`, 2013.

[38] Samuel, M., Saxena, P., and Song, D. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[39] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[40] Saxena, P., Molnar, D., and Livshits, B. ScriptGard: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[41] ScrewTurn Wiki. `http://www.screwturn.eu/`, 2013.

[42] Stamm, S., Sterne, B., and Markham, G. Reining in the Web with Content Security Policy. In *Proceedings of the International World Wide Web Conference (WWW)* (2010).

[43] Su, Z., and Wassermann, G. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)* (2006).

[44] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).

[45] 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 (NDSS)* (2007).

[46] Wassermann, G., and Su, Z. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007).

[47] Weinberger, J., Barth, A., and Song, D. Towards Client-side HTML Security Policies. In *Proceedings of the USENIX Workshop on Hot Topics in Security* (2011).

[48] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., and Song, D. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (Leuven, Belgium, 2011).

[49] Xie, Y., and Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium (USENIX)* (2006).

[50] Yu, F., Alkhalaf, M., and Bultan, T. Stranger: An Automata-based String Analysis Tool for PHP. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2010).