Implementation-level Analysis of the JavaScript Helios Voting Client

Michael Backes*†, Christian Hammer*, David Pfaff*, Malte Skoruppa*
* CISPA, Saarland University
† Max Planck Institute for Software Systems, Saarland University
{backes,hammer,pfaff,skoruppa}@cs.uni-saarland.de

ABSTRACT

We perform the first automated security analysis of the actual JavaScript implementation of the Helios voting client, a state-of-the-art, web-based, open-audit voting system that is continuously being deployed for real-life elections. While its concept has been exhaustively analyzed by the security community, we actively analyze its actual JavaScript *implementation*. Automatically ascertaining the security of a large-scale JavaScript implementation comes with major technical challenges. By creating a sequence of program transformations, we overcome these challenges, thereby making the Helios JavaScript client accessible to existing static analysis techniques. We then automatically analyze the transformed client using graph slicing, reducing an approximately 7 million node graph representing the information flow of the client's implementation to a handful of potentially harmful flows, each individually consisting of less than 40 nodes. Our interpretation of this analysis results in the exposure of two thus far undiscovered vulnerabilities affecting the live version of Helios: a serious cross-site scripting attack leading to arbitrary script execution and a browser-dependent execution path that results in ballots being sent in plaintext. These attacks can be mitigated with minor adaptations to Helios. Moreover, our program transformations result in a version of Helios with fewer external dependencies and, accordingly, a reduced attack surface.

CCS Concepts

•**Security and privacy** → **Information flow control;** *Vulnerability scanners; Web application security;*

Keywords

Information flow analysis; JavaScript; remote electronic voting

1. INTRODUCTION

Electronic voting protocols have received tremendous attention by the scientific community in the last few years. Their appeal and their increased acceptance even for real-life elections are fueled by their ability to offer efficient, sound tallying while providing users the convenience of voting remotely. One of the most widely deployed electronic voting protocols is Helios [9, 10]: a state-of-the-art, web-based, open-audit voting system that has seen real-life deployment in a variety of different settings, such as for the election of a university president [10], student elections [14, 5], as well as for the election of the IACR committee [3].

From a security perspective, remote electronic voting protocols such as Helios typically exhibit a highly complex design that uses advanced cryptographic primitives such as homomorphic encryptions, mixnets, and zero-knowledge proofs, that involves many interactions between different parties, and that intends to achieve a wide range of sophisticated security properties. Consequently, securely designing such protocols constitutes a highly challenging and intriguing task. A multitude of approaches have been recently proposed to automatically ascertain central security properties for electronic voting, such as vote privacy or vote verifiability. In particular the security of the Helios protocol has been thoroughly investigated and its security rigorously proven for many of its intended security properties. As of now, Helios constitutes one of the most widely examined voting protocols in the scientific literature.

Virtually all existing approaches for conforming the security of voting protocols focus on identifying conceptual (logical) or algorithmic (cryptographic) attacks against the protocol considered, i.e., they consider a protocol's symbolic abstraction or algorithmic description, and are therefore agnostic to security violations that arise in the actually deployed implementation. However, history has shown that even security protocols long deemed and even formally proven secure can exhibit severe implementation-level vulnerabilities. Particularly illustrative examples are the recently discovered *Heartbleed* in the OpenSSL cryptography library, or Apple's *goto fail* in its own SSL/TLS implementation. An implementation-level analysis is thus of the utmost importance for every security protocol that should see widespread real-life deployment while intending to offer strong security requirements. Electronic voting protocols, with their strong dependency on societal acceptance, clearly cannot afford severe implementation-level vulnerabilities, and thus naturally call for corresponding analyses.

1.1 Our contribution

We are the first to perform an analysis of the expected security properties of the Helios voting client at the *implementation* level. This goal encompasses several major points.

Security reification through code transformation and static analysis We provide code transformations and static analysis to track potentially harmful information flows that harm the confidentiality and integrity properties in a JavaScript implementation. The transforma-

SAC 2016,April 04 - 08, 2016, Pisa, Italy

ACM 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851800

tions involve the replacement of specific features, whose presence makes reliable static analysis impossible. By replacing these features with functionally equivalent code, we enable existing static analysis techniques and are able to faithfully model the information flow within a complex JavaScript program by a static dependency graph. By phrasing integrity and privacy properties as an information flow problem, we can use graph slicing to significantly reduce the number of nodes under consideration from roughly 7 million to a handful of potentially threatening flows, of which two can be leveraged into real-world exploits. Our approach describes a general means to enable and conduct a security reification through static analysis in real-world JavaScript programs.

Vulnerabilities We report the presence of two vulnerabilities in the JavaScript Helios voting booth client, that, despite years of manual and conceptual analysis, have not yet been revealed: *1*) a cross-site scripting (XSS) attack resulting in *arbitrary script execution*; and *2*) an undocumented feature, which causes the client to send unencrypted plaintext votes without the prior consent of or notification to the user.

Availability of a hardened client The aforementioned code transformations yield two independent benefits. First, they make the voting client amenable to static analysis. Second, they yield an implicitly hardened version of the Helios voting client with fewer external dependencies and thus a reduced attack surface. We make this hardened version publicly and freely available [1]. For the sake of exposition and reproducibility of the individual steps, this version intentionally does not yet integrate the fixes to the vulnerabilities that we report on. We stress that while our analysis was performed on the transformed code, the found vulnerabilities are equally present in the original code.

1.2 Related work

Conceptual attacks on Helios A multitude of approaches have been recently proposed to formalize central security properties for electronic voting (e.g., [11, 18]). The analysis of Helios in particular has received a tremendous attention from the scientific community.

Several publications investigate *privacy* of ballots in Helios, and the notion of *vote independence* has given rise to considerable debate: Vote independence means that by seeing a voter's encrypted ballot, another voter should not be able to cast a meaningfully related ballot. The authors of [16] show that Helios does not satisfy vote independence and exploit this fact in order to compromise vote privacy. They discuss a countermeasure known as *ballot weeding*, and show that their revised scheme offers vote privacy in a symbolic model. The authors of [12] define vote privacy in a computational model and prove that this revised version of Helios fulfills their definition, though only under non-standard assumptions. In parallel, [13] studies pitfalls of the Fiat-Shamir heuristic for non-interactive zero-knowledge proofs used in Helios, and shows that a stronger variant of the heuristic leads to ballot independence.

Helios puts an even greater concern on *verifiability* (both individual and universal) than on privacy, and thus, the extent to which Helios fulfills this expectation has also been thoroughly investigated in the literature. [25] puts forth a formal definition of verifiability in a symbolic model and uses it to analyze the Helios protocol. A more fine-grained model to assess the verifiability of eVoting protocols such as Helios is presented in [26]. They show that Helios is vulnerable to so-called *clash attacks*, wherein malicious administrators could surreptitiously replace a voter's ballot, and discuss countermeasures. In [13], the authors also show how the aforementioned pitfalls of the Fiat-Shamir heuristic may be exploited by colluding election administrators to break universal verifiability in Helios. Finally, [15] defines the notions of *weak* and *strong* verifiability in a computational model. They provide a generic way to transform weakly verifiable election schemes into strongly verifiable ones.

Practical attacks on Helios There is only scant work that considers actual attacks against the implementation of Helios. Instead of exposing flaws in the implementation of Helios itself, some related work has demonstrated exploits in incidental components using Helios as a case study. [19] shows how vulnerabilities in Adobe Reader can be exploited in order to install a malicious browser rootkit that subverts the integrity of a user's vote in Helios. Their attack does not identify a vulnerability in Helios; it is only used as a case study. Similarly, [30] highlights logical web application flaws that arise from using TLS in an insecure manner, and also uses Helios as a case study in order to show how this can be exploited to surreptitiously cast votes on behalf of honest voters. Like our work, [20] focuses on the analysis of the client-side implementation of Helios. However, their code inspection is done purely by hand, and the vulnerabilities they find are different from those that we exhibit.

Static analysis of JavaScript During the last decade, there has been extensive research into information flow violations, which can break the integrity or confidentiality of programs. Early approaches focused on type systems [27], but they tend to be excessively complex and conservative. Jensen et al. model the HTML DOM and Browser API [23] as an extension of earlier work on type analysis [24]. Richards et al. [29] dispel common myths on the use of *eval* in a large-scale study. In turn, [22] shows how *eval* can, in certain cases, be safely removed to aid static analysis. Andromeda [31] delivers demand-driven tracking of potentially vulnerable information flows in JavaScript.

2. THE HELIOS VOTING SYSTEM

The Helios voting system is available in well-documented open source form [2], and a public server is running on http://heliosvoting.org, allowing anyone to create and run their own election. In this paper we focus on the current version (Helios 3.1) in its latest revision (f977ea6586) available at the time of this writing.

The server-side code is mostly implemented in Python (with Django), while the client-side code (which runs in the user's browser) is written in HTML and JavaScript. Our analysis focuses exclusively on the client-side code implementing the voting booth.

2.1 A short review of the Helios protocol

In Helios, any registered user may create a new election. In the initial setup phase, the user who created the election, considered as the *administrator*, can set up the ballot and other election data and specify a list of eligible voters. A key pair is generated by the Helios server, or a set of trustees, for each new election.

Once the administrator is ready, they can *freeze* the election and move on to the submission phase, in which eligible voters may submit ballots. On a high level, the submission phase is very simple: First, a voter requests a specific election from the Helios server, which sends back the browser voting application, called the *voting booth*, as well as the corresponding election data. The voter uses the voting booth to record and encrypt her answers and to submit her encrypted ballot to the Helios server. It is only at this point that the Helios server asks the voter to authenticate herself: By doing so, the voter confirms her wish to actually *cast* her ballot. Finally, the voting server records her encrypted ballot, along with the voter's identity (or an alias) on a public bulletin board.

In the tallying phase, an encrypted tally is computed from all published ballots using homomorphic properties of the encryption

scheme (see [17, 10]), which is then jointly and verifiably decrypted by the trustees. This can be publicly audited by anyone.

In Helios 3.x, authentication in the submission phase can leverage third-party web services such as Google, Facebook or Twitter.

2.2 The Helios voting booth

The most complex element during the submission phase is the voting booth. Its behavior is depicted in Fig. 1. It guides the voter through the questions and records her answers. After her choices have been recorded, the voting booth encrypts the ballot. The voter may then choose to either submit the encrypted ballot to the server or to audit the ballot. In the latter case, she may later submit her ballot after encrypting it with new randomness.

Designed with security in mind, the voting booth is written as a *single-page web application*: After initially pre-loading the election data and page templates, the voting booth makes no further network requests until the ballot is sealed and submitted to the voting server. JavaScript functions implement the entire functionality of the voting booth and take care of updating the rendered HTML user interface during the interaction with the voter. Our aim is to verify that the voting booth fulfills the expected security requirements and that neither its integrity nor its privacy can be compromised. We stress that the threat model here is not a corrupt voter (who could use another application in the first place), but rather a (passive or active) attacker trying to exploit vulnerabilities in the actual voting booth implementation interacting with a honest voter in order to learn, or even surreptitiously modify, a voter's vote. To this end, we analyze the behavior of the actual voting booth's *implementation* using automated tools in order to find potentially hidden flaws. The code analysis is complicated by the fact that it depends on a multitude of complex dependencies and third-party libraries, such as jQuery. We discuss these challenges in depth in the next section.

3. CHALLENGES IN THE ANALYSIS OF THE JAVASCRIPT IMPLEMENTATION

3.1 Implementation-level security properties

Existing approaches that verify security properties for Helios focus on the security protocol on a symbolic or algorithmic level. Thus, they are agnostic to security violations that happen on the application layer. The symbolic primitives are too coarse-grained to capture the minute details of the JavaScript language semantics. Conversely, note that the client merely implements one part of the entire protocol. Therefore, our aim is not to prove the entire protocol secure by formulating some implementation-level properties that the client must uphold. Rather, we want to identify necessary conditions that the implementation must fulfill so that symbolic security properties are not violated by the implementation. We phrase these conditions as information-flow properties: A breach of confidentiality and the associated loss of privacy is perceived as the flow of secret data, e.g., a vote, without proper sanitization (e.g., encryption) to a public channel, such as a network request. A breach of integrity is perceived as the flow of untrusted input to critical data stored on the client side that can force the client to behave in an unintended manner. To analyze the security of the implementation w.r.t. such properties, we leverage static analysis, which derives a semantics-preserving abstraction of the program, such that its (in-)security according to a formal model can be automatically inferred.

3.2 JavaScript is highly dynamic

JavaScript uses higher-order functions and closures, extensive type coercion rules, and a flexible object model where objects can be changed at runtime by adding or removing fields and methods. These dynamic and abstract features encumber static analysis. On top of these language features, ECMAScript consists of 161 functions and objects that need to be modeled. In particular the function *eval* and its variants allow dynamic construction of program code from strings. Reasoning about the behavior of such code requires a-priori knowledge of the strings that can appear and their analysis is therefore not amenable to static analysis. Therefore, highly dynamic features like *eval* need to be removed or their effects conservatively approximated. Fortunately, the core Helios client does not make use of *eval*. Third party libraries, however, do. We consider the difficulties arising in this context in Section 3.4.

3.3 JavaScript cannot be separated from the HTML DOM and browser API

JavaScript programs are usually executed in a rich environment. Web applications execute in a browser environment that interacts with the Document Object Model (DOM) representing the page's HTML, as well as sophisticated libraries such as jQuery. Unfortunately, some of those interfaces (e.g. DOM) are implemented in C++, which prevents fully automatic analysis. Typical client-side programs are thus specified in a combination of scripting, specification and low-level programming languages, for all of which a specification is required.

Execution in JavaScript is *event-driven*; hence the analysis must model the event system, including dynamic registration of event handlers, event bubbling and capturing (recursive triggering of events in nested DOM components) and event-specific object properties. Further, all event handlers are callback functions, i.e., they are queued when a specific event triggers. This leads to asynchronous execution, which results in fragmented code and unstructured execution paths that static analysis must somehow resolve.

The HTML document structure also interferes when resolving variable names defined in HTML attributes. If an event handler is triggered, the scope chain includes *all* DOM objects in the lookup path from the HTML element containing the trigger up to the root of the document. Consider the following example:

```
<script>var src="foo.png";</script>
<img src="bar.png" onclick="alert(src)"/>
```

The onclick-event handler references the variable `src`. However, it is not the variable `src` previously defined in the script tag, but the `src` variable in the `img` tag. Hence the onclick action will trigger an alert containing `bar.png` rather than `foo.png`. Furthermore, the HTML API features a number of non-trivial interactions. For example, setting the `onclick` property of an HTML element at runtime causes a string to be interpreted as event handler code.

3.4 Included libraries

To compound the problems above, many applications build on libraries that alleviate browser incompatibility problems or simplify common tasks such as HTML DOM navigation and AJAX communication. Among others, the Helios voting client uses jQuery, Underscore and class.js, which simulates classical object-oriented programming paradigms including inheritance. From a static analysis perspective, these libraries—while convenient for a programmer—complicate the analysis process severely. By providing their own abstraction *on top* of very abstract features such as event handling and DOM objects, a highly precise modeling of heap structures and a high degree of context- and flow-sensitivity is required in order to produce helpful results with a low number of false alarms. jQuery in particular includes the `$` function that has completely different semantics based on its argument, which can be anything ranging from an HTML string to a DOM element.

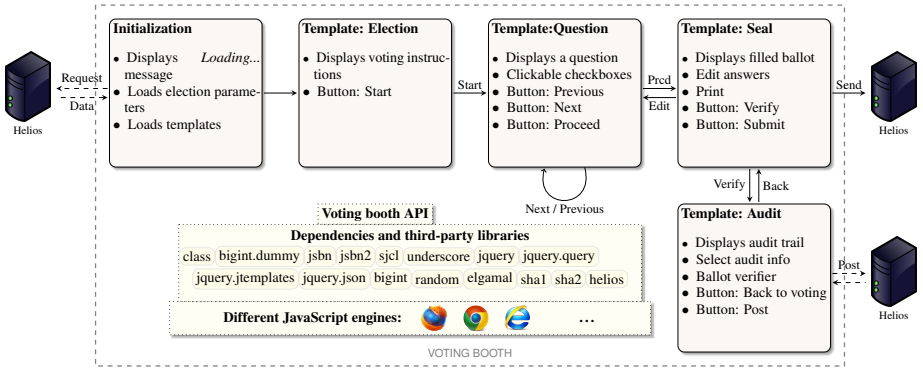

Figure 1: User interaction and implementation of the voting booth

4. IMPLEMENTATION-LEVEL ANALYSIS

In this paper, we focus exclusively on the client-side code implementing the voting booth. The phases of our analysis reflect the challenges explained in the previous section. We present code transformations that resolve the majority of the problems caused by included libraries (see Section 3.4). We then process the results of our transformations with WALA to provide a unified program representation and a number of analyses tackling problems caused by the JavaScript and the HTML/Browser components (see Section 3.2 and Section 3.3). Finally, we automatically look for potentially dangerous paths between manually specified sources and sinks (see Section 3.1). Our findings as well as the applicability of our findings to the original code will be discussed in Section 5.

WALA [7] is a Java library originally designed to provide static analysis capabilities for Java bytecode. It has been used in several research projects as well as further analysis tools, such as Andromeda [31]. Most of the WALA API internally leverages the WALA IR (intermediate representation) instead of source code. The IR is general enough to represent and to analyze other languages as well: Currently there are frontends for Java and JavaScript. Among other analyses, WALA supports call graph construction and pointer analysis, interprocedural dataflow analysis and context-sensitive tabulation-based program slicing [33].

4.1 Code transformations

The implementation of the Helios voting booth heavily relies on the third-party libraries such as jQuery, Underscore and others.[1] Due to the highly reflective nature of these libraries, it is extremely hard to perform automated static analysis on the Helios voting booth's code. Their size is another problem: The uncompressed version of jQuery 1.2.2 (as used by Helios) amounts to 100 KB (its compressed version 60 KB) as compared to about 50 KB for the Helios voting booth (excluding smaller dependencies). While jQuery and other libraries make developing Web applications easier, they typically prevent automated static analysis, as current tools, including WALA, can only cope with some of the dynamic features that are present in these libraries, and even for these only in a very limited way (this is subject to active research as discussed in Section 1.2.) To enable static analysis, we hence refactored the Helios implementation so as to use native JavaScript equivalents. These code transformations yield a client that is independent of the aforementioned libraries. The changes were canonical and could even be refactored automatically.

In this section, we exemplarily describe some of these code transformations. Our complete modified code is publicly available [1].

[1] http://jquery.com, http://underscorejs.org

The modified code is functionally identical to the original code, i.e., the voting booth works in exactly the same way and any changes can be applied to the current version of the complete Helios system.

The jQuery library for JavaScript provides facilities for accessing and updating the DOM, handling events or writing Ajax applications, in a convenient and portable manner. One of its key benefits is that it avoids the need for developers to deal with JavaScript DOM API idiosyncrasies across browsers, and allows them to write concise and legible code. However, newer standards for the browser, like the DOM API, CSS and HTML5, provide equivalent functionality for most of jQuery's APIs, which yields a straightforward refactoring.

Among the core jQuery functions used by Helios are those for performing asynchronous HTTP requests. Namely, Helios uses the `.get()`, `.getJSON()` and `.post()` methods (all of which are wrapper functions for jQuery's `.ajax()` method that sets up a JavaScript `XMLHttpRequest` object). These functions are particularly interesting for our analysis, since they constitute information sinks that may potentially lead to confidential information being sent over the network, as discussed later. The following jQuery call performs an asynchronous HTTP request to `url` and calls the given success handler function upon receiving the server's reply.

```
$.get(url, function(response) { /* process answer */ });
```

In pure JavaScript, this functionality is somewhat more boilerplate:

```
var request = new XMLHttpRequest();
request.open('GET', url, true);
request.onload = function() {
  if (request.status>=200 && request.status<400) {
    var response = request.responseText;
    /* process answer */ }};
request.send();
```

The code for the `.getJSON()` and `.post()` functions is similar.

4.2 Intermediate representation

As noted previously, the intermixing of JavaScript and HTML is commonplace, but unduly hinders static analysis. In order to faithfully process all aspects of such programs, WALA integrates the HTML components into a unified JavaScript model. After conversion, WALA leverages Rhino [6] to parse the JavaScript program, creating an *intermediate representation* (IR). The IR represents a method's instructions in a Java bytecode-like, static single assignment (SSA) form that eliminates stack abstraction and instead maps variables to symbolic registers. As is typical in compilers, the IR organizes instructions in a *control-flow graph*.

4.3 System dependency graphs

As a next step, WALA converts the IR to another program representation more suited to information flow analysis: *system dependency graphs* (SDG). SDGs are used to conservatively approximate all possible information flow within a program. Formally, an SDG $G = (N, E)$ for a program p is a directed graph where the nodes in N represent p's statements and predicates and the edges in E represent the dependencies between them [21]. The SDG is partitioned into *procedure dependence graphs* (PDG) that model single procedures. In a PDG, a node n is *control dependent* on a node m, if m's evaluation controls the execution of n. n is *data dependent* on a node m, if n may use a value computed at m. The PDGs are connected at *call sites*, consisting of a call node c (i.e., a node containing an `invoke` or `dispatch` statement) that is connected with the entry node e of the called function. Parameter passing and result returning, as well as side effects of the called function, are modeled via formal *parameter* and *return* nodes and edges. For passed parameters there exists an appropriate formal node at caller and callee sites, called `PARAM_CALLER` and `PARAM_CALLEE` respectively. Likewise, there exist `RETURN_CALLER` and `RETURN_CALLEE` nodes for return value passing. The `PARAM_CALLER` nodes (referred to as formal-*out* nodes) are control dependent on the calling statement c, whereas the `PARAM_CALLEE` nodes (called formal-*in* nodes) are control dependent on the function entry node e. Likewise for the return nodes. This parameter passing model guarantees that all inter-procedural effects of a function are propagated via call sites. A machine-checked proof [32] shows that the SDG is a conservative approximation to the real data and control flows in a program, i.e., it contains all actual flows. For the running example in Fig. 2, the formal-in and out nodes are constructed as described in the previous paragraph. For the sake of exposition, we append `v1`, highlighting the argument `z`, to the graph.

4.4 Slicing

Slicing [28] is used to find all nodes in the SDG reachable from a specific *seed* node. In the example in Fig. 2, we are interested in which statements can influence the value passed as a parameter to the function `iszero` at a specific call site. We thus compute a *backwards slice* containing only nodes reachable by traversing dependencies backwards, be they control, call or data dependencies. Starting at `v1`, the data dependency can be followed backwards to `PARAM_CALLEE`, from where it passes out of `iszero` back into the calling function to the `PARAM_CALLER` node, and from there to the statement `a = 3`. By also including control dependencies (indirect flow) we can similarly include `iszero` (from `PARAM_CALLEE`), `v3 = invoke iszero a` (from `iszero` and `PARAM_CALLER`) and `foo`.

4.5 Information flow analysis

Conventionally, information flow analysis distinguishes between *explicit* and *implicit* flows. Explicit flows correspond to directly copying secret (high) information to a public (low) variable for confidentiality, and vice-versa for integrity. Implicit flows appear when the *control flow* of the program, i.e., the sequence of statements that are executed, is dependent on high variables.

Numerically, the full SDG of the Helios client consists of roughly 7 million nodes and is therefore impossible to analyze manually. Computing appropriate slices w.r.t. a given information flow concern yields subgraphs of at most 6000 nodes. Only considering paths between high and low statements leaves us with only a handful of different paths containing less than 40 nodes each and therefore suitable for manual inspection.

Confidentiality We state our confidentiality problem as a *declassification* problem: Sending high, confidential data over a low, public channel without declassifying first constitutes a compromise of confidentiality. In the SDG this corresponds to a path from a high input (e.g., a secret vote) to a low output (e.g., an XMLHttpRequest) without a declassification mechanism (e.g., encryption). Note that there is no automatic decision procedure whether a function is an *appropriate* means of declassification.

Fig. 3 shows parts of a slice resulting from slicing backwards from an XMLHttpRequest `send`. The node `dispatch send v50` represents a call to the function `send`, its argument is stored in `v50`. The value `v50` is computed from the previous statement `invoke v52 v4`. Following the data dependencies backwards, one eventually crosses from the callee `ajax_post` into the calling function `request_ballot_encryption`. In this function, we reach the statement `v46 = getfield answers`. This statement retrieves the highly confidential votes. Since this data dependency path contains no declassification, we have to consider it as potentially dangerous. Upon manual inspection, we find that none of the functions constitute an appropriate means of declassification.

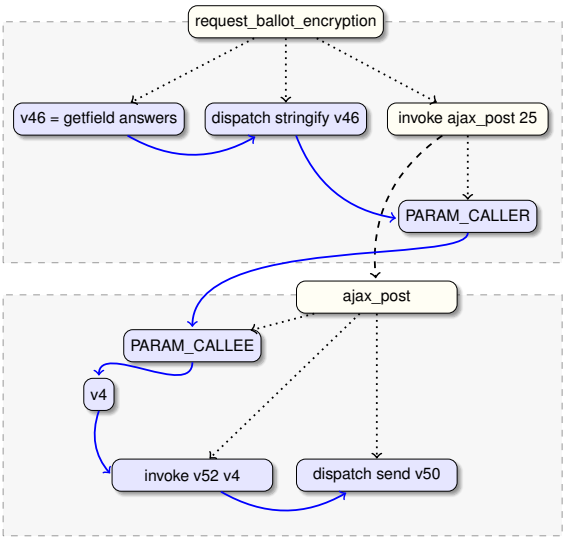

Figure 3: Relevant parts of a backwards slice with `XMLHttpRequest.send()` as the seed.

Integrity Conversely, integrity is formulated as an *endorsment* problem: Public, low input that reaches high-integrity data without prior endorsment compromises integrity. In terms of information flow, we can thus phrase this problem as a forward flow from calls that retrieve low input (e.g., `GET` parameter values) and eventually lead to a high variable (e.g., internal data used to display possible answers or cast a vote) without passing through an endorsement function (realized by sanitizing these parameters). As with declassification, there is no automatic way to ascertain that a function is an *appropriate* means of endorsement. Hence we again require human insight to confirm the functions called on a path are sufficient.

5. VULNERABILITIES

Our automatic analysis found two flaws in the client's code: one breach of integrity, and one breach of confidentiality. We verified that these flows can be exploited in practice in the live version of Helios by successfully deploying corresponding exploits in a mock election. The breach of integrity results in a browser-independent vulnerability leading to arbitrary script execution, while the breach of confidentiality is only evident in a subset of browsers.

```
var foo = function(){
  a = 3;
  b = iszero(a);
};

var iszero = function (z){
    return z == 0;
};
```

```
//foo
a = 3;
v3 = invoke iszero a;
b = v3;

//iszero
v3 = binaryop(eq) v1, 0;
ret v3;
```

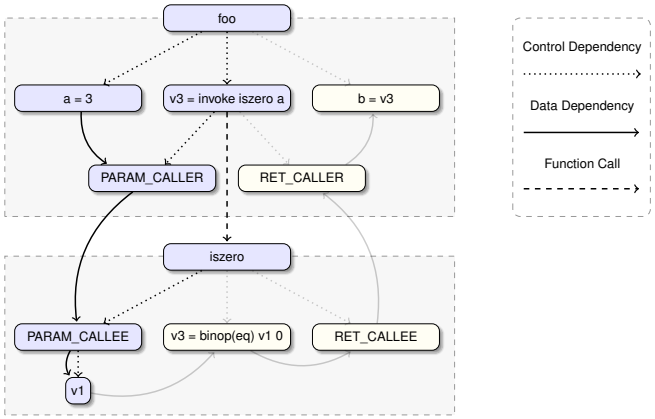

Figure 2: Two functions with their SDG and a backwards slice (blue) of `v1`.

5.1 Arbitrary script execution

The first flaw we discovered is a *cross-site scripting* (XSS) attack. We notified the authors of Helios of this vulnerability and they acknowledged that it is a severe problem that they intend to fix. Cross-site scripting is considered one of the most critical and most prevalent security vulnerabilities in web applications [4]. XSS occurs when poorly validated user input sent to the browser is *executed* by the browser's interpreter, instead of, e.g., being *displayed* as text. The execution of this exploit depends on the attacker being able to deceive the victim into clicking a specially crafted link that directs her to the Helios server.

Reflected client-side XSS exploit In our case, an XSS vulnerability arises from a specially crafted `GET` parameter. The Helios voting booth is generally loaded via a URL such as

```
http://heliosvoting.org/booth/vote.html?election_url=
    /helios/elections/<UUID>
```

where `<UUID>` is an election-specific identifier. The `GET` parameter `election_url` is processed by the client-side code.[2] The client parses this parameter in order to load election data, election metadata, and additional entropy from the server:

```
var election_url = $.query.get('election_url');
...
$.get(election_url, function(resp) {
    /* set up election data */ });
$.getJSON(election_url+"/meta", function(resp){
    /* set up election metadata */ });
$.get(election_url+"/get-rand", function(resp){
    /* add server randomness */ });
```

With a proper parameter `election_url`, these requests invoke the Helios server API to return JSON objects containing data to initialize the booth. Unfortunately, the parameter `election_url` is not sanitized on the client side. Hence an attacker can point it towards an external resource, e.g., an attacker-controlled server:

```
http://heliosvoting.org/booth/vote.html?election_url=
    http://evil.com/get-bad-data
```

The server at `evil.com` is set up to return corrupted JSON strings.

Circumventing the same-origin policy Note that this will not work if done naively, as the browser's *same-origin policy* would prevent processing the retrieved JSON strings: The requests will be sent, but the response will be `undefined` in the JavaScript code. However, web servers may allow their replies to be processed by client-side scripts running on other domains using the *cross-origin resource sharing* (CORS) mechanism. That is, the attacker can set up her server so as to attach the following HTTP headers to the replies containg the corrupted JSON strings:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: X-Requested-With, Origin,
    Content-Type, Accept
Content-Type: application/json
```

The responses will then be successfully passed to and processed by the corresponding handler functions. Using this approach, the attacker can manipulate the election data contained in returned JSON strings, with severe consequences: The attacker can compromise the integrity of the vote by intentionally mislabeling the answers (e.g., switching the displayed order of names) in a vote, deceiving the user into voting for the wrong person. Likewise, the attacker can violate vote privacy by modifying the encryption key used to submit the encrypted ballot. The attack even deceives the ballot auditing process: The verification link generated by the voting booth contains the same `GET` parameter as the URL of the voting booth. Still, while the attacker can alter JSON object-specific values to her liking, she is still unable to execute arbitrary code on her own.

Arbitrary script execution The attack can be escalated even further in the original client by setting up an external server that sends JavaScript instead of the expected JSON object. This leads to the script being loaded and executed by the Helios voting booth client. This behavior is a consequence of how jQuery evaluates the `$.getJSON` function when it is called with an external URL as its argument: It tries to circumvent the same-origin policy by itself. Instead of issuing an XMLHttpRequest for the resource, jQuery creates a `<script>` tag inside the DOM's header and sets its `src` attribute to the remote URL. Since remote scripts included in this manner are intentionally exempt from the same-origin policy, this causes the browser to load and execute the retrieved content. Normally, `$.getJSON` would expect to retrieve a JSON object, which does not hold executable content. By sending a JavaScript instead of a JSON object from the remote server, the attacker gains the ability to execute arbitrary code.

5.2 Leaking the vote

Additionally, we found a flaw in a program path that leads to an *unencrypted ballot* being openly sent over the network. Specifically, when the voting booth's DOM is loaded, the Helios client confirms that the client supports *web workers* (scripts running in background threads). Web workers are used in Helios to perform

[2]The parameter is presented in URL-decoded form for readability.

encryption of a voter's choice and generation of zero-knowledge proofs. Surprisingly, if the browser does not support web workers, Helios simply requests the server to encrypt the ballot, and sends it the unencrypted ballot:

```
$.post(BOOTH.election_url + "/encrypt-ballot", {'answers'
    : $.toJSON(BOOTH.ballot.answers)}, function(result)
    { /* process encrypted ballot */ });
```

Clearly, this code was included on purpose, yet this comes as a complete surprise, as the voting booth does perform a network request while interacting with the voter, contradicting the premise of a single-page web application and the claim of the original paper [9]. Sending the plaintext ballot violates all assumptions. It could be argued that using a secured HTTPS connection, a passive attacker cannot read the secret ballot; however: This also means that an additional layer of encryption on top of Helios' own encryption is needed to guarantee privacy. This is not mentioned in the original paper [9], leading one to believe the Helios protocol by itself already guarantees privacy. Furthermore, the version of Helios downloaded from the original Github repository contains installation instructions that lead to the Helios client running over plain HTTP by default. While it is possible to run the Helios server-side implementation on top of a TLS terminator, this requires additional knowledge and expertise. The documentation neither explains the procedure nor mentions its necessity. Lastly, even when run over HTTPS, the key pair used for TLS is a different one than the key pair generated by Helios for each election. In particular, an administrator with access to the server's private HTTPS key can compromise vote privacy of voters who use older browsers—even in an election set up with an election key pair generated by a set of trustees.

In summary, running the Helios client in browsers that do not support web workers leads to a clear violation of vote privacy. Browsers that do no support web workers, yet see a non-trivial deployment in practice include Internet Explorer 9 and earlier, as well as all available versions of Opera Mini.[3] Depending on the actual statistics used, the combined worldwide percentage of people using these browsers lies between 12% and 36%.[4] A secure way to deal with these browsers would be to simply disallow them completely and prompt the voter to select a different browser. At the very least, this unexpected behavior should be clearly documented and plainly pointed out to election administrators. Currently, neither is the case. When we notified the Helios authors of this vulnerability, they stated that they were not concerned, since in Helios, some inherent trust is placed in the server anyway. While they acknowledged that the claim of a "single-page web application" from the first paper is no longer true, they argued that the alternative of not supporting old browsers is unacceptable. They also pointed out that, even though the election server may indeed see a large portion of plaintext ballots during the election process, there is no single-owner *long-term* storage of plaintext ballots. In summary, their point of view is that the need for usability and the support of a wide range of versions of all major browsers outweighs the threat to voter privacy by a possibly malicious server administrator.

6. DISCUSSION AND TAKEAWAYS

Although our analysis was performed on a modified version of the Helios client, the attacks also apply to the original client. We confirmed these vulnerabilities in an unmodified Helios client. The converse does not hold in general: We saw in Section 5.1 how an attack that allowed arbitrary modification of a set of variables could be amplified to arbitrary script execution in the original client. This was possible due to jQuery's internal behavior, and does not apply to the transformed code. Hence, our analysis on the transformed code is mostly useful to uncover previously unknown vulnerabilities, but not as a constructive proof of security.

The original Helios paper [9] expected auditors to closely investigate the client-side JavaScript code, and that using the jQuery library would make it easier to understand and analyze the implementation due to its abstraction layer on top of low-level JavaScript functionalities, which makes the code more concise and easier to follow. As actual auditors of the code, we found that this is not necessarily the case. From a developer's perspective, modern browsers are more compatible than ever: Standardized and well-documented APIs for DOM traversal and manipulation, event handling or server communication have been adopted by all major browsers, decreasing the demand for jQuery [8]. While it might be argued that jQuery eases support for older web browsers such as Internet Explorer 9 or earlier, we saw in Section 5.2 that supporting such browsers may induce additional vulnerabilities jQuery cannot prevent. From an analyst's perspective, automated analysis becomes much harder in the presence of libraries, while manual analysis implies putting blind trust in the security and behavior of a third-party library.

We conclude that a voting booth that does *not* rely on jQuery is easier to inspect and trust. Most of the functionality provided by jQuery can be implemented in native JavaScript code that runs in all modern browsers, removing a highly complex library. The same applies for the Underscore library and the implementation of class inheritance in JavaScript. This also significantly reduced the program size: With all uncompressed libraries included, the original version of the Helios client code amounts to almost 500 KB and over 9000 LOC in total. Without these libraries, the client has less than 250 KB and under 4000 LOC. Keeping dependencies low is a good idea both for security reasons and conciseness of the entire codebase. Note that the code transformations that we implemented to simulate the functionality of third-party libraries used by Helios can easily be exported into an external lightweight library. This allows our analysis to be easily reproduced in future versions of Helios. We note that at the time of publication of the Helios paper at USENIX 2008, compatibility between browsers was a greater issue and that these libraries made a lot more sense. Due to the rapid development of browsers, performance of modern JavaScript engines and the availability of standardized, cross-platform JavaScript APIs, the client codebase could be drastically reduced, easing code review processes and increasing trust in its implementation.

Finally, we point out that the vulnerabilities that we found are easily fixed. The confidentiality problem (Section 5.2) is a purposefully built (though questionable) feature that can be removed. For the integrity problem (Section 5.1), it suffices to sanitize the parameter obtained from the URL query string. We stress the fact that although these attacks are simple, no manual code review has unveiled them so far, which highlights the benefits of an automated analysis. In fact, a very similar bug had already been reported,[5] yet has not led to the discovery of the XSS vulnerability described in this paper.

7. CONCLUSION

We performed the first *implementation-level* analysis of the Helios JavaScript voting client—one of the most widely deployed and analyzed remote electronic voting protocols—and showed how to overcome the intricate technical challenges associated with analyzing a real-world JavaScript web application. By faithfully modeling and subsequently reducing a 7 million node dependency graph to a handful of potentially harmful information flows, we discovered two

[3] http://caniuse.com/#feat=webworkers

[4] http://gs.statcounter.com, http://netmarketshare.com

[5] https://github.com/benadida/helios-server/issues/41

new, heretofore unknown, vulnerabilities: a major XSS vulnerability that was escalated to arbitrary script execution, and a minor flaw that led to leaking the plaintext ballot.

Our methodology addressed the gap between real-world and statically analyzable code, and we expect approaches in the same vein can be applied in a variety of related settings to find vulnerabilities in client implementations that were overlooked during manual audits.

Acknowledgments

We thank the anonymous reviewers for their comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the initiative for excellence of the German federal government.

8. REFERENCES

[1] Hardened Helios Javascript client. https://infsec.cs.uni-saarland.de/~skoruppa/sac2016/heliosbooth_sac.zip

[2] Helios Github. https://github.com/benadida/helios-server.

[3] Should the IACR use e-voting for its elections? http://iacr.org/elections/eVoting/.

[4] The open web application security project. https://owasp.org.

[5] Princeton Helios voting. https://princeton.heliosvoting.org/.

[6] Rhino. https://developer.mozilla.org/en/docs/Rhino.

[7] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.

[8] You might not need jQuery. http://youmightnotneedjquery.com.

[9] B. Adida. Helios: Web-based open-audit voting. In *Proc. 17th USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

[10] B. Adida, O. de Marneffe, O. Pereira, and J. Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of Helios. In *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '09*. USENIX Association, 2009.

[11] M. Backes, C. Hritcu, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proc. 21st IEEE Computer Security Foundations Symposium, CSF 2008*, pages 195–209. IEEE Computer Society, 2008.

[12] D. Bernhard, V. Cortier, O. Pereira, B. Smyth, and B. Warinschi. Adapting Helios for provable ballot privacy. In *Proc. 16th European Symposium on Research in Computer Security*, pages 335–354. Springer, 2011.

[13] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to Helios. In *Proc. 18th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT 2012*, pages 626–643. Springer, 2012.

[14] P. Bulens, D. Giry, and O. Pereira. Running mixnet-based elections with Helios. In *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '11*. USENIX Association, 2011.

[15] V. Cortier, D. Galindo, S. Glondu, and M. Izabachène. Election verifiability for Helios under weaker trust assumptions. In *Proc. 19th European Symposium on Research in Computer Security*, pages 327–344. Springer, 2014.

[16] V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. In *Proc. 24th IEEE Computer Security Foundations Symposium, CSF 2011*, pages 297–311. IEEE Computer Society, 2011.

[17] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proc. International Conference on the Theory and Application of Cryptographic Techniques*, pages 103–118. Springer, 1997.

[18] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[19] S. Estehghari and Y. Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking Helios 2.0 as an example. In *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '10*. USENIX Association, 2010.

[20] M. Heiderich, T. Frosch, M. Niemietz, and J. Schwenk. The bug that made me president a browser- and web-security case study on Helios voting. In *E-Voting and Identity - Third International Conference*, pages 89–103. Springer, 2011.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[22] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA 2012*, pages 34–44. ACM, 2012.

[23] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011.

[24] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Static Analysis*, pages 238–255. Springer, 2009.

[25] S. Kremer, M. Ryan, and B. Smyth. Election verifiability in electronic voting protocols. In *Proc. 15th European Symposium on Research in Computer Security, ESORICS 2010*, pages 389–404. Springer, 2010.

[26] R. Küsters, T. Truderung, and A. Vogt. Clash attacks on the verifiability of e-voting systems. In *IEEE Symposium on Security and Privacy, SP 2012*, pages 395–409. IEEE Computer Society, 2012.

[27] A. C. Myers. jFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 228–241. ACM, 1999.

[28] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. *Speeding up slicing*, volume 19. ACM, 1994.

[29] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *25th European Conference on Object Oriented Programming*, pages 52–78. Springer, 2011.

[30] B. Smyth and A. Pironti. Truncating TLS connections to violate beliefs in web applications. In *7th USENIX Workshop on Offensive Technologies*. USENIX Association, 2013.

[31] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Proc. FASE 2013*, pages 210–225. Springer, 2013.

[32] D. Wasserrab and D. Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop, VERIFY 2010*, pages 141–155, 2010.

[33] M. D. Weiser. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. 1979.