12-2016

Context-Sensitive Auto-Sanitization for PHP

Jared M. Smith
University of Tennessee, Knoxville / Oak Ridge National Laboratory, jms@vols.utk.edu

Richard J. Connor
University of Tennessee, Knoxville, rconnor6@vols.utk.edu

David P. Cunningham
University of Tennessee, Knoxville, davpcunn@vols.utk.edu

Kyle G. Bashour
University of Tennessee, Knoxville, kbashour@vols.utk.edu

Walter T. Work
University of Tennessee, Knoxville, wwork@vols.utk.edu

Recommended Citation

Smith, Jared M.; Connor, Richard J.; Cunningham, David P.; Bashour, Kyle G.; and Work, Walter T., "Context-Sensitive Auto-Sanitization for PHP" (2016). *University of Tennessee Honors Thesis Projects.*
https://trace.tennessee.edu/utk_chanhonoproj/2138

Context-Sensitive Auto-Sanitization for PHP

Jared Smith (Lead), Kyle Bashour, Joseph Connor, David Cunningham, Travis Work

Team 9 • Cisco Systems, Inc. • April 26th, 2016

Table of Contents

Note: The numbering of the contents follows the final report rubric from the course instructors.

3.0. Executive Summary 3
3.1. Project Goals 3
3.2. Summary of the Results 4
3.3. Challenges and Outcomes 4
4.0. Requirements 6
4.1. Platform 6
4.2. Sanitization Support 6
4.3. Additional Features 6
4.4. Initial Candidates for Testing 7
4.5. Performance Target 7
4.6. Taint Sources 7
4.7. Untainting 8
5.0. Change Log 8
5.1. Chronological List of Agreed Upon Changes to the Requirements 8
6.0. Documentation of Design Process 8
6.1. Design Chosen Based on the Requirements 8
6.1.1. Decomposing the design based on the requirements 8
6.1.2. Open questions answered by research 14
6.1.3. Alternative solutions explored 15
6.1.4. Selected solution 16
6.1.5. Ensuring the requirements were met 17
6.1.6. Validation of work against previous phases 17
6.1.7. Results and outcome 19
7.0. Lessons Learned 21
7.1. What Went Wrong 21
7.2. What Should Be Done Differently in Future Work 22
8.0 Statement of Team Members' Contributions 24
8.1. Team Roles 24
8.2. Contributions of Each Team Member 24
9.0. Signatures of all Team Members and Customer 26
9.1. Names, Signatures, and Dates 26
9.2. Dissenting Statements (signed) – If Any 27

3.0. Executive Summary

3.1. Project Goals

Cross-Site Scripting (XSS) vulnerabilities are a class of web application vulnerabilities that allow an attacker to take control of a victim's web browser's interaction with a vulnerable web application. XSS was found to be over 60% likely to be an exploitable vulnerability on websites using PHP as the primary server-side language[1]. XSS vulnerabilities can also leave significant damage if exploited. Attackers leverage XSS vulnerabilities to take over user accounts (at times gaining administrative access), commit financial fraud, steal private data, deface popular websites, etc. In 2011, Mike Samuel from Google and several other researchers described how Context-Sensitive Auto-Sanitization can be used to automatically prevent XSS vulnerabilities. Context-Sensitive Auto-Sanitization (or CSAS) augments the type system of web template languages to automatically detect and prevent XSS vulnerabilities. The technique has been successfully employed in a number of template languages, including: Closure Templates for Java and JavaScript, CTemplate for C and C++, AngularJS for JavaScript, and html/template for Go. One notable exception is PHP.

PHP is an extremely popular language for web application development with a large base of existing code, composing roughly 82% of public websites[2]. Unfortunately, PHP's built-in preventions for XSS vulnerabilities are primitive and cumbersome to use. The programmer must manually invoke PHP's sanitizers whenever potentially malicious data is sent to the browser making it trivial to create errors of omission. Worse yet, even when faithfully applied, PHP's

[1] http://www.slideshare.net/jeremiahgrossman/whitehat-security-website-security-statistics-report-q109

[2] http://w3techs.com/technologies/overview/programming_language/all

built-in sanitizers do not prevent all XSS vulnerabilities. With sponsorship from Cisco Systems, Inc., we are bringing context-sensitive auto-sanitization to PHP.

3.2. Summary of the Results

We developed PHP-CSAS as an extension to the PHP language, in order to prevent XSS vulnerabilities in the language. Our extension optimally prevents several major types of XSS attacks against the PHP language and ultimately PHP applications. It is unobtrusive: it works in the background so the web developer doesn't have to. It is flexible: adding more sanitizers for future attacks is painless using our rich documentation. It is fast: it operates with performance comparable to PHP applications without our extension with less than 1.9x overhead. We built an extensive testing suite composed of passing tests for all overridden PHP functions, from network and file I/O functions to database queries, the included sanitization functions, and the routines for safe and unsafe tracking of data. Additionally, we built a robust demonstration framework and included demonstration sites made of representative PHP web applications for showing the effectiveness of our extension against XSS. From the use of our demonstration and the indication from our testing suite, it is clear that PHP-CSAS accomplishes the goal of repelling XSS with little to no developer effort required apart from installing our extension.

3.3. Challenges and Outcomes

We faced several obstacles along the way, from minor to major implementation challenges that we had to overcome to issues with designing a test suite and demonstration that would cover as many edge cases as possible while still being feasible to accomplish in the given time frame. Despite the obstacles we encountered, we overcame them by working together as a highly skilled team of engineers that stayed in constant communication in order to stay in sync on our progress at every step of the way. Ultimately, PHP-CSAS is functional, robust, and easily extensible, and can serve as at least a proof-of-concept that context-sensitive auto-sanitization in PHP is achievable with minimal overhead and maximum effectiveness. For production environments, PHP-CSAS will work as expected, but work can always continue to be done on the framework, which is why the team and customer have decided to make the project open source. This will help bring in the broader community of PHP developers and security experts in order to advance the current state of web security to new levels with a framework built for stopping XSS in the web's most popular server-side language.

4.0. Requirements

4.1. Platform

4.1.1. The modification must run on Apache 2.4.18 with mod_php on Ubuntu Server 14.04 LTS.

4.1.1.1. Additional versions of Apache may be explored as time permits.

4.1.1.2. Additional Server environments (BSD, Solaris, etc.) may be explored as time permits.

4.1.2. The only guaranteed to be supported version is PHP 5.4.

4.1.3. Any further PHP versions that will be supported will be provided as time permits. 4.1.4. The modification must be relatively easy to install.

4.2. Sanitization Support

4.2.1. Must support HTML PCDATA

4.2.2. Must support Quoted HTML Attributes

4.2.3. Must support Unquoted HTML Attributes

4.2.4. Must support URL Start

4.2.5. Must support URL Query

4.2.6. Must support URL General

4.2.7. Must support JavaScript String Value

4.3. Additional Features

4.3.1. Must have the Ability to register additional PHP functions as sanitizers for supported contexts.

4.4. Initial Candidates for Testing

4.4.1. Must test with Wordpress

4.4.2. Must test with MediaWiki

4.4.3. Must test with Roundcube Webmail

4.5. Performance Target

4.5.1. Must work with no more than a 4x slowdown in page render time for a representative benchmark.

4.6. Taint Sources

4.6.1. The system must have the ability to taint input from the following components of HTTP requests:

4.6.1.1. Must work with GET

4.6.1.2. Must work with POST

4.6.1.3. Must work with Cookies

4.6.1.4. Must work with Headers

4.6.2. The system must have the ability to taint input from file I/O.

4.6.3. The system must have the ability to taint input from network I/O.

4.6.4. The system must have the ability to taint input from database queries.

4.6.4.1. Must work with PDO Queries

4.6.4.2. Must work with MySQLi Queries

4.6.5. The system must have the ability to taint input from the process environment.

4.7. Untainting

4.7.1. The system must have the ability to untaint a given data source or value that is known to be trusted.

4.7.2. The system must have the ability to mark a given data source or value as already sufficiently sanitized for a given context.

5.0. Change Log

5.1. Chronological List of Agreed Upon Changes to the Requirements

- **February 19th, 2016 - 12:37 PM:**

  Changed guarantee of compatibility with PHP 5.2 to PHP 5.4 due to Apache 2.4 and Ubuntu 14.04 being incompatible with PHP 5.2. Affects requirements number 4.1.2 above.

6.0. Documentation of Design Process

6.1. Design Chosen Based on the Requirements

6.1.1. Decomposing the design based on the requirements

We begin with a high level description of the design of the application. There are five major components in the application: the module for hooking into the functions of PHP that outputs data and capturing that output to pass to our extension, the module for tracking whether input is safe or unsafe for any given context, the HTML parser for getting contexts from input, sanitizers to escape the different contexts that are marked as unsafe, and the module for taking the sanitized output and passing it back into the PHP application where we initially pulled it from (to be output to the browser at this point).

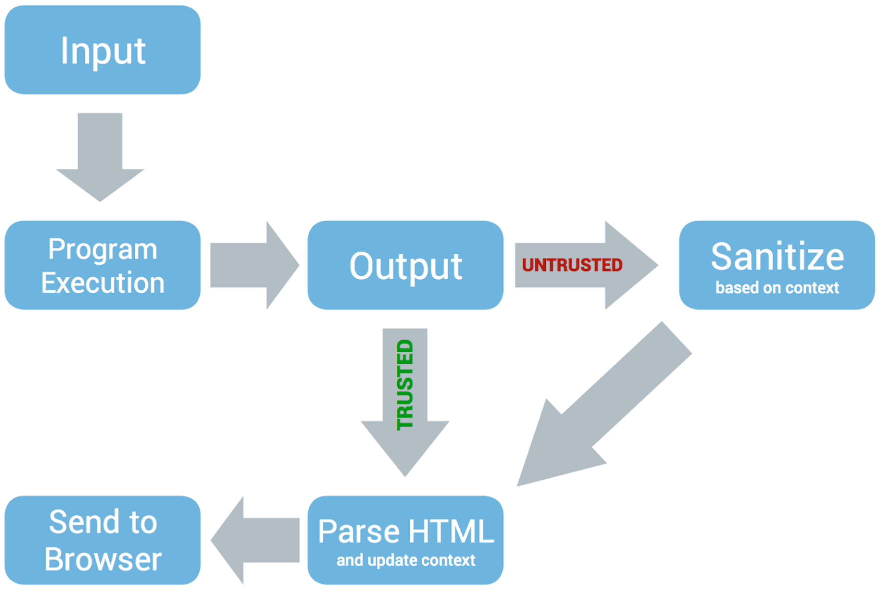
The first component and the second component are encapsulated into a pre-existing PHP extension called Taint. This extension detects potentially unsanitized input, such as tainted strings, from user input passed into the application. It also has the ability to spot SQL injection vulnerabilities and file inclusion vulnerabilities among other features. When taint is enabled, it will warn the user whenever a tainted string is passed through into the program logic through the standard methods of input in PHP: GET and POST parameters, cookies, database queries, file and network I/O, etc. We used this extension to fulfill the first two steps of sanitization which are to first hook into the output from the PHP core and then to be able to track safe or unsafe data for given contexts. These first two components interface with each other through standard function calls in our code, and once the data has been determined to be safe or unsafe, it is ready to be passed to the sanitizers after being parsed by our HTML parser to determine context of the state of the input into the application. Several challenges we have identified with this step were the following: could we integrate the existing Taint code as the base of our extension, does Taint hook into all possible places where PHP can output data, and can we modify Taint to track unsafe and safe data for data objects other than strings (such as objects). We were able to do all three with the existing Taint extension and successive modifications to it, notably the ability to mark objects as unsafe.

The third component of the application, which has output passed to it from the the Taint extension that marks data as safe or unsafe, is the HTML parser. We have borrowed most of the HTML parser code from CTemplate, a template language by Google for C that already has auto-escaping functionality built in. This component uses a state machine to track contexts of data flowing through the application, which we call in our “echo”, “print”, and other handlers on every passed input from Taint in order to do the following: know the context of input at any given time in order to choose the correct sanitizer to use, and to update the context after each input has been processed by the sanitizers so that the next piece of input is in the correct context. We did not face any significant challenges here other than the efforts to translate the C++ interface to the HTML parser in the CTemplate code into a C interface in PHP-CSAS. However, it should be noted that the core of the HTML parser borrowed from CTemplate was already in C, so integration was rather trivial once compiler errors from the integration were resolved.

The fourth component of the extension is the robust set of sanitizers for taking given input and an associated context from the third component (the HTML parser), determining the appropriate sanitization function for the given context, passing the unsanitized input to the sanitization function if and only if the input is tagged as unsafe for the given context from its pass through the second component (the Taint tracker), and then receiving it back from the sanitization function with the input properly escaped. Once this final step is complete, this component passes the data to the fifth and final component to be given back to the PHP core and eventually to the browser. Since Taint already warns on output, we hijacked the warning from Taint as simply a marker of whether input is safe or unsafe, and we passed that along to the logic to determine if something needs to be sanitized as described above. Input will only be sanitized if and only if the input is marked as unsafe from Taint. This defines how we leverage the existing Taint codebase into our application. The major problem we had to solve here was determining how to obtain a string representation of the input passed into our extension from the core. This was an issue because PHP turns all objects into an underlying representation in the language called "zvals" and we needed a string representation of the zvals in order to properly sanitize. This problem was solved by diving into the PHP core source code and finding a function of the

PHP API for making a “printable” zval. We couldn’t find any relevant documentation on this function or the API in general, as it largely does not exist in the community. In fact, the only documentation on the PHP API that we could find was incomplete, even the documentation for the API on the official website admitted that it is not fully documented. Unfortunately, for much of this project we had to dive into the PHP source code and reverse engineer the language to understand how we should interact with the underlying undocumented PHP API.

The fifth and final component of the application handles passing the sanitized input back to the PHP core to be sent to the browser. An illustration of the extension’s semantics can be seen below encapsulating the entirety of the previous five steps. Though the final step is partially handled by the existing Taint extension, we had to modify the pipeline to ensure that the properties of the data we were modifying were not altered after sanitization, except for removing malicious characters. For example, we had to ensure that the length of a string before sanitization in the string’s length property was the same after sanitization as before, so that the semantics of PHP functions are not altered through sanitization in a way that a typical PHP developer would be unfamiliar with. Another potential problem we faced was the issue of determining the best method for modifying a zval before it is output. We built a proof of concept to test our solution to this and determined that the input does in fact retain its prior properties after modification during output; therefore, we did not have an issue with this.



Additional challenges that we had to address but developed solutions for were: would we need to override any functions that have already been overridden by taint or other extensions and how would we deal with that, how would the taint extension and sanitizers interact, what would we do when an application manually sanitized untrusted data correctly, and what would we do when an application manually sanitizes untrusted data incorrectly. The answers to these questions were as follows: we did not have to worry about overriding other functions that Taint overrides because Taint and our extension will not be running concurrently, we did not concern ourselves with how our extension interacts with other extensions because as far as we were concerned PHP-CSAS is an independent extension that simply sanitizes input passed into the application (however, we did put disclaimers into the documentation that state that this extension directly modifies input to the application and developers should be aware of this when using PHP-CSAS), we did modify Taint to use our sanitizers instead of simply producing a warning as we discussed before, when an application manually sanitizes data correctly our extension detects this and does not over-sanitize the data, and when an application manually sanitizes data for an incorrect context we also detected this and sanitized it again for the correct context.

Additionally, we designed a way to reproduce development environments. We chose Vagrant[3] to help in the automation of creating reproducible developer environments, which allowed us to share one script through version control that will setup an entire environment for writing the extension. This enables all the core members of the development team to be working off of the same environment as everyone else. We also used version control through Git and GitHub extensively to stay in sync as a development team. Over the time of development, we ended up with multiple Vagrant scripts, each for new versions of the demonstration and development environment. For example, when we determined we would need MySQL in our environment, we simply modified the existing vagrant script to encompass that installation process.

We also built an extensive test suite to ensure our extension covers all possible cases of XSS injection. Our test suited allowed us to make sure that the extension would not slow an application running PHP-CSAS down more than four times its original speed. We built our test suite on top of the Taint extension's existing test suite. For ensuring that our extension worked with the third-party popular PHP applications we stated in the requirements, we tested individual implementations of the latest versions of these framework with our extension and ensured they worked correctly including proper sanitization. These third-party applications were Wordpress, MediaWiki, and Roundcube Webmail.

[3] https://www.vagrantup.com/

Lastly, we designed a robust demonstration to present at the senior design presentation days. We developed three identical sites apart from three distinct properties: a site that uses neither PHP-CSAS or htmlspecialchars to sanitize data, a site that uses only htmlspecialchars to sanitize, and a site that only uses PHP-CSAS to automatically sanitize data. With these sites, we were able to test that our extension correctly sanitized input for a representative web application in PHP.

6.1.2. Open questions answered by research

We answered several problems in addition to the ones presented above through research completed by our team. We had to determine whether Taint handled all foreseeable forms of input, and whether we needed to add the additional injection points specified in the requirements document. We determined that Taint did not handle all foreseeable forms of output and places where data could be executed or included in the browser while untrusted. Some places that were handled included handler functions implemented in Taint already like “echo”, “print”, “print_r”, “sprintf”, “eval”, “include”, “require”, “include_once”, “exec”, “system”, and “shell_exec” among others. We had to add many more overridden functions like “file_put_contents”, “printf”, “vprintf”, “mysqli_query” (and other “mysqli” functions), “oci_parse”, “passthru”, “proc_open “require_once”, PDO functions (the other MySQL database library included in PHP), and input that is concatenated through the “+” operator in PHP.

We also had to modify the Taint functionality to treat all non-string objects as unsafe for any context so that we are sure to sanitize them correctly. To do this, we marked all string properties of the object, and all object and array properties of the initial object unsafe in a recursive manner. This ensured that any possible data that could come from that object into the browser would be appropriately sanitized when it is accessed inside the object and run through one of the overridden functions, like “print” or “echo”.

Another problem we had to solve involved making sure data wasn’t over-sanitized. In order to ensure compatibility with existing applications, without significant changes to the existing application, when the application correctly sanitizes data, it should not be over-sanitized (e.g. HTML escaped twice when it should have been escaped once). Conversely, when the app sanitizes data incorrectly for the output context, auto-sanitization should sanitize it correctly. To solve this issue, we had to modify existing sanitizer functions so that their return value is marked safe for the appropriate context and the data won't be over sanitized.

6.1.3. Alternative solutions explored

When deciding on the best solution for this problem, we had to consider many different possible problems associated with developing a secure extension to prevent XSS attacks. Our final product needed to prevent all types of XSS attacks without adding more than 4x overhead. Our solution also needed to work across multiple platforms. It had to be easy to install on platforms such as Wordpress, MediaWiki, and Roundcube Webmail. We also needed our prototype to be easily testable, because we needed to be able to secure PHP from all XSS attacks, which would require an extensive test suite to guarantee that XSS is blocked with the extension installed.

We did look extensively into the possibility of modifying the PHP core. In order to correctly modify the PHP core, we would have to take all of the previously mentioned possible problems and account for them. First, the advantage of modifying the PHP core would be that we would not be limited in the amount of code we could modify. This is an advantage as well as a disadvantage because if we dived into the core and modified too much, we may cause serious slow-downs, or possibly change semantics without intending to and creating more vulnerabilities. Modifying the PHP core would also make testing our solution very time consuming. It would be difficult to know exactly what the cause of an overhead gain would be if we modified the core. Whereas, testing an extension, could be easily implemented with tests that would be trivial to write given the experience of our team. Additionally, modifying the PHP core for this particular version of PHP might not be compatible with future editions of PHP, and there is also a strict approval process from the core PHP developers to modify the language itself, which involves an often time-consuming RFC process that has to approved by the majority of the core PHP developers[4]. Essentially, the large overhead of modifying the PHP core, along with its limited testability and improbability of working with future versions are why we decided to go against this possible solution.

6.1.4. Selected solution

After considering all of the requirements of our finished product, writing an extension rather than modifying the core emerged as the clear winner. There are many well documented extensions already being used by web applications, even if many of these extensions don't dive into the language as much as ours. The amount of documentation on how to develop and implement an extension contributed to us deciding on this solution. Another advantage of writing an extension is that it will be more lightweight and easier to test than if we modified the PHP core. Typically, well-developed extensions are also compatible with the existing platforms that we need to run on. This will allow us to fulfill the requirement of an easy install and implementation

[4] https://wiki.php.net/rfc

on all our required programs such as Wordpress, MediaWiki, and Roundcube Webmail. From our research, it was determined that writing an extension that sanitizes output and then outputs that sanitized data back to the PHP application will allow us to prevent XSS attacks against PHP applications.

6.1.5. Ensuring the requirements were met

To ensure that our extension meets all the requirements, many different tests had to be passed. First, we had to finish our extension's implementation and ensure that our test suite covered all of the use cases and was passing. After we did this we had to come up with a way to test it for speed and efficiency. In order to do this, we knew we would need to use a PHP profiler. Therefore, we chose xdebug, xhprof, and webgrind, all of which form a PHP application profile when combined. For the testing, we had to write tests for each of our sanitizers individually. We also needed to test that the additional input and output functions we added (for network I/O and file I/O) that we are interfacing with, and to test whether we are able to mark data as safe or unsafe. We tested PHP-CSAS against several popular web platforms written in PHP, notably Wordpress, MediaWiki, and Roundcube Webmail with others in line as time permitted.

6.1.6. Validation of work against previous phases

At the requirements phase of our project we met and communicated with our customer to ensure that we understood thoroughly all of the requirements our project must meet. Several of the agreed on requirements included; a performance parameter, easy implementation on popular PHP frameworks, and the ability to register additional PHP functions as sanitizers for supported contexts. The remaining requirements are in section three of this document. We tried to consider all of the problems we may run into with the requirements, but ultimately felt that they were fair parameters for this project. From there, we proceeded to design the selected solution.

For the design portion of the project, we met many times as a team to design our solution. We carefully considered multiple ways to solve this problem. Almost immediately it was obvious to us that we would either have to modify the PHP core or develop an extension. We had to consider the advantages and disadvantages of both possible solutions. After we did sufficient research we decided that developing an extension rather than modify the PHP core would be the best way to meet all of our requirements. These are the steps we took during the design phase of the project in order to make sure that our design would be verified against the outcomes of the prior phases. Once we had a plan for our design, we started implementing our plan.

The implementation phase of our project consisted of developing the extension. This was the most extensive part of the project. We had to first make sure that the taint extension was properly used. Taint is an existing extension that inspects PHP for malicious user input[5]. Integrating Taint with our design plan was a very important part of our project. This pre-existing extension was the first part of our implementation, and getting it to work correctly reassured us that our previous two phases would yield good results. After integrating Taint, we began working on the HTML parser and sanitization functions. After writing the sanitizer and manually inspecting them, we wrote the additional input and output functions we needed to override according to the requirements document, notably database functions and network and file I/O which was not done by Taint. Throughout the development of these functions, we constantly debugged and tested our code to make sure there were no errors. Even with all of this going on in the implementation phase of the project, we still had to ensure that our implementation was

[5] http://php.net/manual/en/book.taint.php

verifying the outcomes of the design and requirements phases. This led us to needing us to develop a way to test our extension against our original requirements.

The testing phase of our project was a very crucial element in making sure we have an effective finished product. While testing our implementation, we tested our sanitizers to make sure they did what they were supposed to. We had to test each sanitizer individually in order to make sure they all worked by themselves before wrapping it all in one extension. For testing the sanitizers and the overridden functions, we used phpt[6], a PHP unit testing framework We also had to come up with a way to test against our performance requirement. This requirement stated that our solution could not slow down the PHP application by more than four times the original speed. For doing the performance testing, we used Xdebug, xhprof, and webgrind. Testing ultimately allowed us to make sure that our implementation, design and requirements were all met.

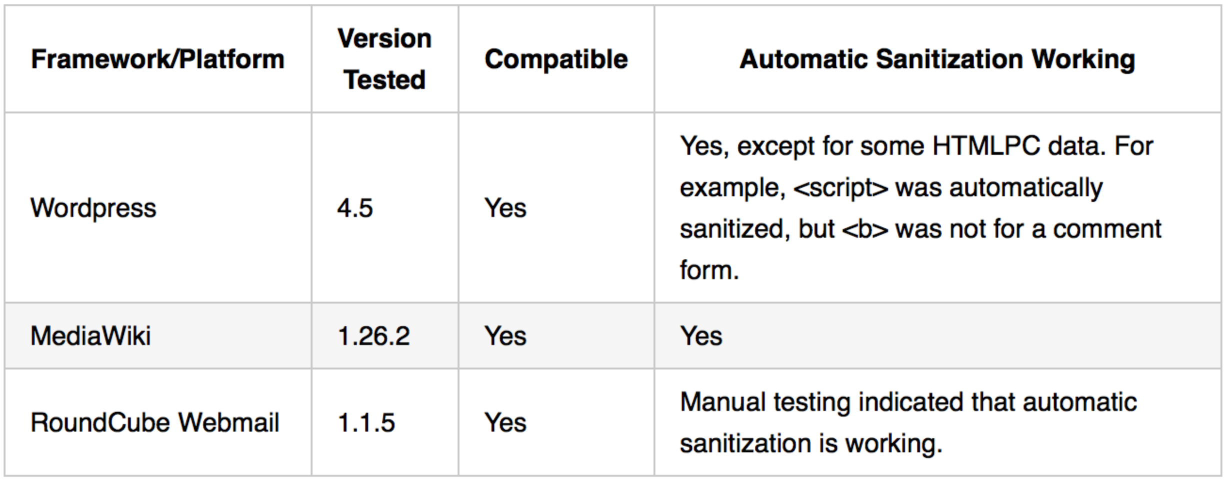
Lastly, our evaluation showed us that we did in fact meet all of the original requirements. All of our tests on the individual sanitizers and the overall speed yielded desirable results, with under 4x overhead. We focused throughout this project on making sure that in each phase we coincided with our original requirements. This allowed us to troubleshoot our extension along the way to ensure our final results would be up to the required standards.

6.1.7. Results and outcome

All tests we have written are passing according to the specifications set in the requirements document. These include tests for the following: I/O functions that need data to be marked safe and unsafe for certain contexts (i.e. printf, echo, sockets, etc.), for the taint tracking

[6] https://qa.php.net/write-test.php

of unsafe and safe data, and the sanitizers. We have finished the performance tests and are under 4x overhead at about 1.9x overhead with PHP-CSAS. These metrics were taken for a realistic PHP application running the extension using the demonstration we developed. Our requirements stated that we must test PHP-CSAS against Wordpress, MediaWiki, and RoundCube Webmail. The compatibility testing for there are pictured below:



Framework/Platform	Version Tested	Compatible	Automatic Sanitization Working
Wordpress	4.5	Yes	Yes, except for some HTMLPC data. For example, <script> was automatically sanitized, but <b> was not for a comment form.
MediaWiki	1.26.2	Yes	Yes
RoundCube Webmail	1.1.5	Yes	Manual testing indicated that automatic sanitization is working.

Wordpress is compatible except for small cases where the platform trusts some user data. MediaWiki functions with PHP-CSAS enabled as well and the auto-sanitization is able to sanitize data from the platform when the manual sanitization is removed. Roundcube Webmail gave us issues trying to install due to not being able to use MySQL as the database, which was an issue not related to PHP-CSAS. However, it was working to the point where we could browse through the webserver UI and test for sanitization. Finally, the overhead in the testing of these platforms was not noticeable over installs of the platforms without PHP-CSAS enabled.

The results from the development and testing of the demonstration clearly showed the effect CSAS has on an identical site apart from three distinct properties: a site that uses neither PHP-CSAS or htmlspecialchars to sanitize data, a site that uses only htmlspecialchars to sanitize, and a site that only uses PHP-CSAS to automatically sanitize data. We designed a PHP website that takes user input in various ways and ensures that all the contexts covered in the requirements document have user input put into them and into there we insert our XSS payloads. The demonstration site uses CSS Bootstrap[7] in order to have a presentable design. We also wrote an automation framework using Python, Vagrant, and the Selenium web driver[8] to automatically open browsers for the identical demonstration site for each of the three properties mentioned above that will type in malicious input to the websites and carry out XSS attacks. We have tested our demonstration for the case of PHP-CSAS enabled and no manual sanitization and it blocks all XSS from affecting the page based on the contexts that were required by the requirements document. The other two demonstration setups fail to block all cases of XSS from affecting the page, effectively proving the success of our extension.

7.0. Lessons Learned

7.1. What Went Wrong

Overall, there could have been improvements in our process and how our team accomplished the project; however, there were few obstacles that took a non-trivial amount of effort to overcome, and given the talent of our team, we assessed the issues and solved them promptly. Notably, several of the open questions we had to answer during the implementation phase caused our chief architect, Joseph, to question the development practices of the PHP core developers. The source of the PHP language contains remarkably poorly written C code in the places we had to interface with, and there were several times that in order to solve the issues we faced with hooking some function in the source from our extension we were baffled for hours on

[7] http://getbootstrap.com/
[8] https://github.com/seleniumhq/selenium

how to decipher all the layers of C macros that the PHP source code uses. On the management side of the project, communication could have been better in the beginning of the project and we could have started working on the implementation earlier. Regarding communication, we started out with using Slack[9] (it was what we used in COSC 401) and it was very reliable and in general the team checked it; however, we switched to HipChat[10] near the beginning and from then on the client itself lead to issues with communication and team members being notified on time. It is worth noting that the switch to HipChat was due to internal team member requests. However, this did not cause serious issues and we were able to function well together as before. Additionally, we could have used more time for implementation, so that we could improve the extension even further. For example, we could have spent time ensuring that substring tracking was done, despite that not being in the requirements. However, as the entire team is composed of seniors with full course loads and part-time research positions or internships during school, there was little time to work on this project unless we moved other engagements out of the way. Ultimately, we ended up working together between 6-8 hours on weekends as a team and between 8-10 hours over the course of every week remotely not including class time where we were required to attend guest lectures or give several presentations.

7.2. What Should Be Done Differently in Future Work

There are several key improvements that could have been made to both our work on the project and to the course structure in general. Regarding the project work, we could have improved the following: used Slack instead of HipChat or pushed the team to check their notifications more often, met more often during the first half of the semester, and started the

[9] https://slack.com/
[10] https://hipchat.com/

project during COSC 401 in the Fall of 2015 in order to allow more time to work. Regarding the course structure, as a team we could have been far more productive if we were allowed to use the class time as mandatory working time except for the days of the team presentations and ethics exams as many of the guest lectures were unanimously familiar content to our team. The material covered in these guest lectures was often either material we already knew or had experienced through internships over the course of our undergraduate careers. Despite these improvements that could be made, we managed to accomplish all of the requirements and successfully deliver a well-tested, demonstrable, proof-of-concept implementation of the proposal envisioned by our customer.

8.0 Statement of Team Members' Contributions

8.1. Team Roles

- **Jared Smith** – Team Leader, Supporting Software Engineer for Implementation, Testing, and Demonstration, Supporting Documenter, Researcher
- **Joseph Connor** – Chief Architect, Lead Software Engineer for Implementation, Supporting Researcher, Supporting Software Engineer for Testing and Demonstration
- **David Cunningham** – Lead Software Engineer for Testing, Software Engineer for Implementation, Researcher and Documenter
- **Kyle Bashour** – Lead Software Engineer for Demonstration/Evaluation, Documentation Support, Researcher and Documenter
- **Travis Work** – Lead Researcher, Documentation Lead, Scribe

8.2. Contributions of Each Team Member

- **Jared Smith** – Overall leadership and management of team, interfaced directly with customer and course instructors on behalf of team, responsible for implementing parts of the implementation, testing, and demonstration, including the automated demonstration framework, reviewed documentation and reports, contributed to research alongside Travis, managed Trello and HipChat for effective task management and team communication.

- **Joseph Connor** – Responsible for the majority of the design and development of implementation, assisted others in researching, counsel for the demonstration and testing, assisted Jared in determining the next steps to take the team on every step of the development process.
- **David Cunningham** – Built and tested the extensive testing suite for the project, responsible for a substantial part of the implementation working with Joseph and Jared, assisted in documentation of the project, and contributed to research alongside Travis and Jared for the selected solution.
- **Kyle Bashour** – Built the three demonstration sites used with the automated framework for testing the effectiveness of PHP-CSAS, contributed the majority of the documentation for the project, assisted in researching the possible solutions from which we chose, assisted research efforts in general.
- **Travis Work** – Contributed to the original research of possible solutions. He also worked with the team on the design portion of the project. Travis wrote most of the team's papers and made sure they were done in a timely manner. Contributed documentation to extension.

9.0. Signatures of all Team Members and Customer

9.1. Names, Signatures, and Dates

Team Members

Jared M. Smith

Signature: Jared Smith Date: April 26th, 2016

Kyle Bashour

Signature: Kyle Bashour Date: April 26th, 2016

Joseph Connor

Signature: Joseph Connor Date: April 26th, 2016

David Cunningham

Signature: David Cunningham Date: April 26th, 2016

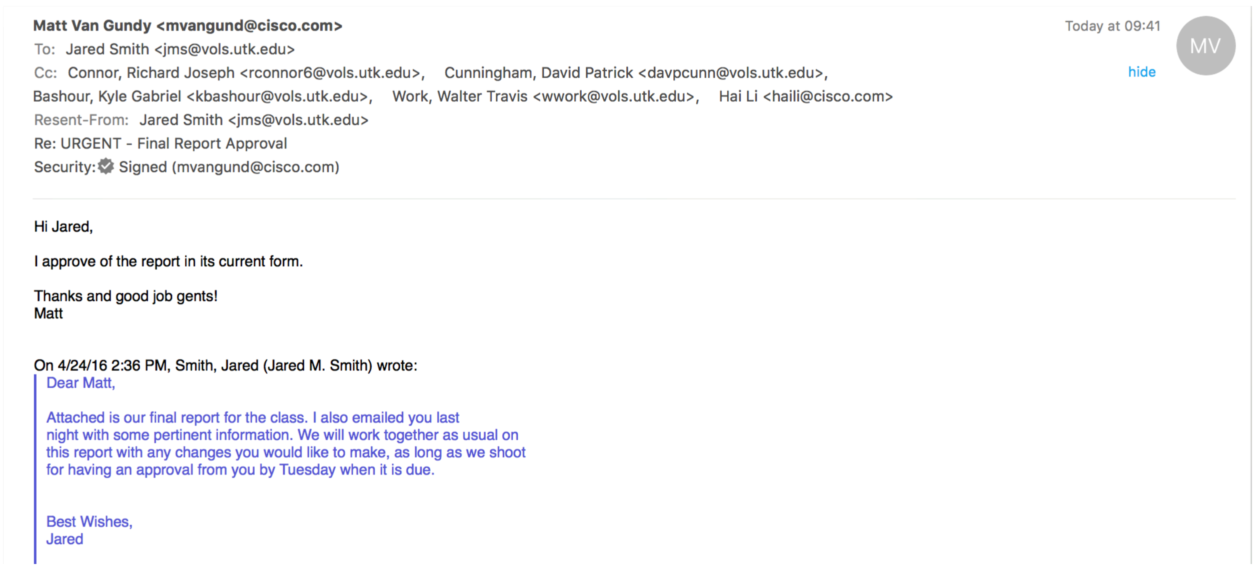
Travis Work

Signature: Travis Work Date: April 26th, 2016

Customer

Matthew Van Gundy - Cisco Systems, Inc.

Approval through Email (See below) Date: April 26th, 2016



Matt Van Gundy <mvangund@cisco.com>
Today at 09:41
MV
To: Jared Smith <jms@vols.utk.edu>
Cc: Connor, Richard Joseph <rconnor6@vols.utk.edu>, Cunningham, David Patrick <davpcunn@vols.utk.edu>, Bashour, Kyle Gabriel <kbashour@vols.utk.edu>, Work, Walter Travis <wwork@vols.utk.edu>, Hai Li <haili@cisco.com>
hide
Resent-From: Jared Smith <jms@vols.utk.edu>
Re: URGENT - Final Report Approval
Security: Signed (mvangund@cisco.com)

Hi Jared,

I approve of the report in its current form.

Thanks and good job gents!
Matt

On 4/24/16 2:36 PM, Smith, Jared (Jared M. Smith) wrote:

> Dear Matt,
>
> Attached is our final report for the class. I also emailed you last night with some pertinent information. We will work together as usual on this report with any changes you would like to make, as long as we shoot for having an approval from you by Tuesday when it is due.
>
> Best Wishes,
> Jared

9.2. Dissenting Statements (signed) – If Any