# emphaSSL: Towards Emphasis as a Mechanism to Harden Networking Security in Android Apps

Xuetao Wei[†], Michael Wolf[†], Lei Guo[*], Kyu Hyung Lee[‡], Ming-Chun Huang[°], and Nan Niu[†]
[†]University of Cincinnati [*]Northeastern University [‡]University of Georgia [°]Case Western Reserve University

***Abstract*—The use of secure HTTP calls is a first and critical step toward securing the Android application data when the app interacts with the Internet. However, one of the major causes for the unencrypted communication is app developer's errors or ignorance. Could the paradigm of literally repetitive and ineffective emphasis shift towards emphasis as a mechanism? This paper introduces emphaSSL, a simple, practical and readily-deployable way to harden networking security in Android applications. Our emphaSSL could guide app developer's security development decisions via real-time feedback, informative warnings and suggestions. At its core of emphaSSL, we use a set of rigorous security rules, which are obtained through an in-depth SSL/TLS security analysis based on security requirements engineering techniques. We implement emphaSSL via the PMD and evaluate it against 75 open-source Android applications. Our results show that emphaSSL is effective at detecting security violations in HTTPS calls with a very low false positive rate, around 2%. Furthermore, we identified 164 substantial SSL mistakes in these testing apps, 40% of which are potentially vulnerable to man-in-the-middle attacks. In each of these instances, the vulnerabilities could be quickly resolved with the assistance of our highlighting messages in emphaSSL. Upon notifying developers of our findings in their applications, we received positive responses and interest in this approach.**

## I. Introduction

### A. Motivation

Though Android platform is the most popular platform in the world, the authorship of Android applications (apps) has been notably insecure [1]–[3]. In the appified world, Android apps faced many challenges in the safe implementation of security. Beyond malware and theft, the use of smartphones on insecure wireless access points remains as a dangerous reality in day-to-day data consumption. Previous studies have shown that holes exist in a wide range of popular Android applications and libraries, including banking applications, e-commerce libraries used to conduct transactions through third-party servers, and social media clients that are shown to be vulnerable [4]–[6]. These holes leave the end-user without the benefits of the HTTPS (Hypertext Transfer Protocol Secure) protocol. However, this basic encrypted security is a reasonable expectation of all non-trivial applications.

The causes for the disparity between the theoretical and realized security of SSL implementations can be attributed to a number of factors including library vulnerabilities, complications and shortcomings of the TLS protocol itself, server misconfiguration, and end users [5]. However, the most pervasive reason why these implementations are not effective is due to application developer's mistakes when making network calls to Internet web servers. Mistakes in code can be caused by leftover debug code or testing-time workarounds such as custom trust managers and skipping hostname verification [4]. Overrides of security errors are also pervasive. Developers have been observed to use code snippets in order to clear error messages without resolving the certificate trust issue [4], [7]. Furthermore, in the case of developers using 'http://' when the 'https://' version of a web service is readily available, the mistake may be caused by developer ignorance, apathy, or frustration with HTTPS-style transport-layer security. Concerns for application speed, functionality, and interoperability with legacy systems may also play a role in the decision to leave applications without the highest layer of protection [5]. Rather than literally repetitive and ineffective emphasis [7], could our focus shift towards emphasis as a mechanism? A more informative methodology to aid in adding security considerations in Android apps is desired. Therefore, it is imperative for developers of HTTPS-ready apps to seek a solution that serves as an assistant in the proper creation of SSL connections with more emphasis. Approaching the problem from this developer-centric position presented challenging issues in the mechanism for determining the most pressing security issues, properly identifying these issues before compile-time, and determining the most prudent phrasing of error messages.

### B. Contribution

In this paper, we present emphaSSL, a simple, practical and readily-deployable way to solve this problem via real-time feedback, informative warnings and suggestions. Through real-time highlighting of unsafe methods, the developer could correct their errors once they understand the implications. Our emphaSSL directly enables developers to fix the mistakes of HTTPS development before compiling their app code. At its core of emphaSSL, a set of rules are utilized, which are derived from the in-depth analysis of security requirements engineering techniques on Android apps. This includes a process of determining user and system assets, application functions, threats, and limitations to a secure environment. This foundation gives our ruleset validity and focus. We design and implement emphaSSL via the PMD source-code parsing project [8], which would serve to point out insecure source code to developers as they write their applications. We evaluate this design through a combination of automated testing and manual review of the test results. Our ruleset's results were verified for accuracy through an investigation of the tool's

output. We provide context to the ruleset by reading over the offending lines of code by hand and comparing them to security best practices. The evaluation results show our emphaSSL is effective at determining user errors with a very low false positive rate (around 2%) before an application is compiled, which ultimately hardens the networking security for Android apps in the wild.

Specifically, we make the following contributions:

- We propose a simple, practical, and readily-deloyable approach to combating developer misuse of Internet-based security connections based on source-code parsing and coding-time feedback. Our approach works with developers to fix issues immediately. By simply installing emphaSSL, developers can begin catching their mistakes. This design is also useful in applications beyond the scope of HTTPS-specific security and can be extended to include future guidelines of next-generation protocols and concerns. Furthermore, this work can be applied to numerous environments.
- We provide practical rules to assist Android app developers in the proper creation of SSL connections. These rules are informed by our design for text-parsing violation checking and were produced through a process rooted in security requirements engineering. The objective of emphaSSL is to verify that major HTTPS overrides are not in place within the application source code. This ruleset includes checks for the use of HTTPS, integrity of hostname verification, security of trust management, and transparency of error reporting.
- We perform a well-proportioned study of 75 open source Android applications that use the INTERNET permission totaling over 100,000 lines of code, which demonstrates the effectiveness of our ruleset with a very low false positive rate, around 2%. Furthermore, we find that 30 of these applications can be considered vulnerable and nine that have serious HTTPS overriding capabilities that leave these applications open to man-in-the-middle attacks. We present a discussion of developer response to our follow-up notifications on their project issues. We find several cases of application owners taking our suggestions seriously and amending their code to properly conform to HTTPS standards.

## II. Overview of emphaSSL

Our emphaSSL, as shown in Figure 1, is an early-warning approach to Android application security. In our approach, we attempt to educate the developer in a noninvasive way on the basics of HTTPS on Android, and solve for security errors in real-time. Instead of relying on external organizations or install-time static-code scanners, we present the belief that developer will fix their mistakes in the implementation of HTTPS if given the proper understanding of the protocol and its implications. To achieve this goal, we present the developer with helpful messages in their development environment when an unsafe method or string literal has been detected. These

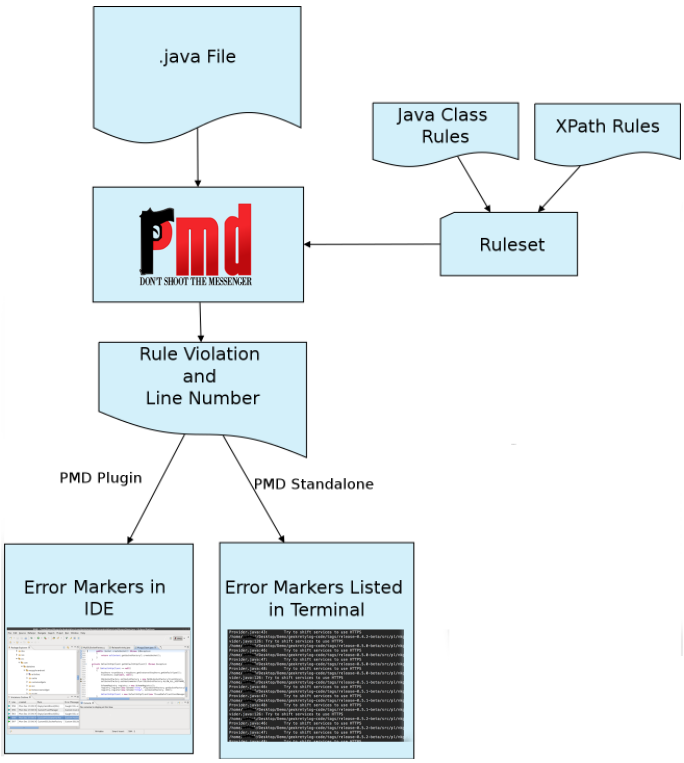


Fig. 1. The architecture of our emphaSSL

messages briefly explain the flaw which was discovered and then present a link to further reading and remediation.

At its core of emphaSSL, a ruleset is built on top of the open-source PMD source-code parsing project [8]. PMD allows for custom rules to be applied to the project as seen in Figure 1. The ruleset for emphaSSL is derived from a process of security requirements engineering and common mistakes discovered by the Android security community. These mistakes are generally exhibited by unsafe methods or literals which compromise the identity validation system of HTTPS. The ruleset of emphaSSL is then compared against source code by PMD either printing out a list of errors into the terminal or placing a marker next to the line of code which has violated a rule in the IDE.

Due to the enormous combinations of Java source code methods, variables, and logic, the ruleset we present with emphaSSL is not all-inclusive. Thus, emphaSSL is designed as an extensible, but still substantial implementation of our design for authorship-time code validation can be effective in networking security environments. In this paper, we test the effectiveness of emphaSSL through automation and manual analysis of a substantial number of applications. This ruleset is effective to guide developers to write safer HTTPS calls by running this ruleset against the application. We will explain our approach, design, implementation and evaluation in the following sections with more depth.

## III. Design of SSL/TLS Security Rules

In this section, we will discuss how our sample rules were chosen to be included in the final ruleset by explaining the way we have set benchmarks for the Android/server PKI system in a security context. Following this section, we will discuss the ruleset we have developed and how they correspond to the design layed out in this section. The goal of the `emphaSSL` is to provide a utility to code authors which analyzes source code in real-time and provides feedback on the security of HTTPS implementation. Our hypothesis for this project was to show that SSL/TLS security errors can be effectively presented and mitigated in a developer's programming environment.

Before deciding on the practical system to use, we first define rules which should not be violated in order to comply with TLS standards. Our design methodology is instructed by the field of security requirements engineering [9]–[13]. We identify the assets, functional requirements, security goals and threats, security requirements, and finally the limitations of these requirements. This `emphaSSL` ruleset design is built on both standard Android HTTP use and the existing HTTPS security system. The goal in either case is to move each application toward a more protected state of either system and, if possible, an HTTPS Everywhere paradigm [14].

**User Assets**: In the scope of this paper, the ruleset is devised to protect assets which are transmitted over the Internet from mobile devices. Thus, these assets include the encrypted (HTTPS) and plaintext (HTTP) messages which are transmitted from Android devices. This message may contain personally identifiable information, login credentials, or innocuous data. The user's device and website logins can also be considered assets as they may become vulnerable if a successful man-in-the-middle attack is carried out.

**Functional Requirements**: The functional requirements which are needed in this mobile data model in order to give the assets value are the ability to connect to a server as well as send and receive messages from the server. Our design must make sure not to compromise the basic operations of an application for sake of extra security. This can functionally take place with either HTTP or HTTPS. Overall, a secure HTTPS connection should be carried out in order to maximize both functionality and security [5], [7].

**Security Goals and Threats**: The primary goals of transport-layer security are confidentiality and integrity. Note that, this work is aimed at moving Internet communications toward HTTPS Everywhere and thus any instance where a message can be encrypted and retain a trusted identity, it should be encrypted and encrypted correctly. Man-in-the-middle attacks can sniff packets, manipulate data, and generally deny service to the user. These threats can fall under a breach of privacy, loss of data, and denial of service, etc. Our ruleset should be able to guide developers to defend against interception, surveillance, and censorship. When correctly implemented, HTTPS prevents man-in-the-middle threats from posing a threat to user data, but this ruleset is designed with an understanding that HTTPS is often insecurely used or simply unused. Our design calls for technical information and its corresponding implications to be presented directly to the developer.

**Security Requirements**: We must determine the components within HTTPS and specifically Java's handling of HTTPS which must be kept together. We identify four sections of HTTPS use which must be required: 1) use of HTTPS, 2) hostname verification 3) valid error checking and 4) proper trust management. These categories then include specific methods and code practices which have been identified as common mistakes made by developers [4]. The use of the `AllowAllHostnameVerifier` would be an example of a violation of the second requirement, hostname verification.

**Limitations**: Given the innumerable ways to violate the four aforementioned aspects of HTTPS, the sample ruleset which we present is practical, thorough and extensive, but not a completely comprehensive checklist. Furthermore, analyzing source code does not have context and may yield false positives when certain code is correctly implemented, but using a blacklisted method. This requires trust in the developer to appropriately interpret the error message and respond in a way that furthers the application's security.

## IV. Development of the emphaSSL Ruleset

Our sample rules follow the security requirements (SR) which were laid out in the previous section. Our `emphaSSL` contains ten rules listed in Table I. This table lists the rule number, the name we gave to the rule which summarizes the error caught, and a short message that appears to the developer if the rule has been triggered on their code. Here, source code of the app will be analyzed during the coding time. Note that, the error messages have been truncated and links to online material have been replaced with 'HYPERLINK'. These ten blacklisted method calls provide a selection of mistakes made in applications using HTTPS or HTTP. The choice of these rules come from our focus on security requirements engineering as well as our understanding of common Android HTTPS vulnerabilities [4], [5] and the Android Developer Training [15]. The rules are written to be as specific and contextually appropriate as possible. Furthermore, our design can be seen as highlighting the 'blindspots' [16]. By choosing commonly misused lines of code, we are primarily identifying mistakes made by developers who are unaware of the issues in security they are causing.

In the following section, we will describe why we chose these rules and why we chose PMD as the host of this plugin extension. In the next section, we will describe the experimental process itself.

**SR. 1 Use of HTTPS**: In order to detect whether or not developers were utilizing HTTPS, we created two rules which serve the similar purpose of identifying whether HTTP URLs are being used in an application. In checking literals, some URLs may resolve when an 's' is added to the URL, in others it may not. For this security requirement, we developed a rule for both situations. These rules could later be weighted based on urgency. A literal that can be changed immediately is more pressing than a literal that cannot change without breaking

| Rule # | Rule Name | Truncated Alert Message |
|---|---|---|
| 1, 8 | AllowAllHostnames | AllowAllHostnameVerifier used. Any certificate received may in fact be from an attacker in between the user and server leaving the user vulnerable to data loss and surveillance. Read more: HYPERLINK |
| 2 | CustomHostnameVerifier | Custom HostnameVerifier used. Custom HostnameVerifiers may overlook certain protections which are applied by default to HTTPS libraries. Read more: HYPERLINK |
| 3 | CustomSSLSocketFactory | Custom SSLSocketFactory used. This opens the user to man-in-the-middle-attacks which compromise data and privacy. Use the library defaults if at all possible. Read more: HYPERLINK |
| 4 | CustomTrustManager | Custom trust manager used. This accepts certificates which are invalid and thus giving man-in-the-middle machines the ability to intercept traffic between the user and server. Read more: HYPERLINK |
| 5 | MoveToHTTPS | While this literal does not resolve to a website when entered as a URL, this connection can be made safer by moving servers and the applications which connect to them to an HTTPS Everywhere paradigm. |
| 6 | HTTPSCouldBeUsed | This literal resolves to an address online when parsed as a URL. By changing the 'http' to 'https', the message will be encrypted as it travels to and from the server. |
| 7 | SSLSetVerifyResult | Unsafe method SSL_set_verify_result used. This method overrides certificate checking in HTTPS, leaving the message vulnerable to interception and sniffing attacks. |
| 9 | EmptyCatchBlock(SSL) | Caught SSL or Certificate exception not reported. An HTTPS-related error was discovered and no actions were taken. This suppresses the error which should be sent to the user in some form notifying them of the unsafe certificate. |
| 10 | UnsafeHostnameVerification | SSLSocket created without proper hostname verification. In order for an SSLSocket to be secure, it needs manual hostname verification connected to it. This prevents sniffing and interception of packets by attackers. Read more: HYPERLINK |

TABLE I
RULESET USED. NOTE THAT RULE 1 AND 8 ARE GROUPED TOGETHER SINCE THEY ARE TWO DIFFERENT WAYS OF IMPLEMENTING THE SAME ALLOWALLHOSTNAME OVERRIDE

functionality until a server is properly configured for HTTPS. Rule 5, MoveToHTTPS, would be triggered if the application is using an HTTP URL, but the URL does not resolve to an appropriate address. Since increasing security in this case would directly break functionality, this rule would not result in a heavy warning, but rather a suggestion for a shift to secure servers if this is possible. Rule 6, HTTPSCouldBeUsed functions in a similar way, but would be violated if the HTTP URL, when changed to HTTPS, resolved to a valid webpage. This would prompt the developer with a more firm warning. In either case, the developer is made aware of the status of their application's Internet security and is shown steps on how to harden the connection.

**SR. 2 Hostname Verification**: The issues which comprise hostname verification overrides are the most common type of HTTPS misuse due to the ease of implementation, but they are also some of the most dangerous overrides in Android security. Several suspect methods and practices have been identified in regards to the overriding of hostname verification, thus three fifths of the rules we have determined fall under this category. Rules 1 and 8 would be violated if a project implements a form of allowing all hostnames. This is achieved programmatically by way of a method or boolean that always returns `true` placed into the hostname verifier or `SSLSocketFactory` . One specific way to implement these ALLOW_ALL verifiers, is to declare a custom hostname verifier with the ALLOW_ALL item as a parameter. Customization of hostname verification is dangerous, potentially allowing for the ALLOW_ALL methods and thus we have listed CustomHostnameVerifier in our rules. If the OpenSSL [17] library is used, the `ssl_set_verify_result` would be used to a similar dangerous result as Rule 1 and 2. Rule 3, CustomSSLSocketFactory, would catch a declared `SSLSocketFactory` . Using a manually crafted `SSLSocketFactory` allows for the customization of the SSL connection and can be used to insert a custom hostname verifier or a foreign trust store. Finally, the direct creation of the SSLSocket does not include hostname verification by default and thus when hostname verification is not provided, Rule 10 would be thrown.

**SR. 3 Valid Error Checking**: The suppression of errors makes the difference between applications that notify the user if they want to continue after detecting certificate mismatches and applications which override certificate verification altogether. The former results in the most secure application a developer can present without compromising functionality while the latter sacrifices all protections offered through HTTPS. Rule 9 EmptyCatchBlock(SSL) most specifically applies to this category. This rule would be triggered when either the catch in a try-catch statement is left blank or the 'throws' component of a method declaration is left empty. This rule checks for specific SSL/TLS-related errors in order to remain relevant to Android SSL/TLS security, but the practice of overriding any error should never be used in a published application. Errors should always be presented to the user.

**SR. 4 Proper Trust Management**: While it is impossible by Java code analysis to detect a self-signed certificate added to the package, an altered `TrustManager` can be detected. This is the purpose of Rule 4, CustomTrustManager, which would be encountered if this component is declared in code. Generally the addition of extra certificates has been seen as insecure even when being included by manufacturers and telecommu-

nication companies [18]. Thus, developers should be warned that adding additional certificates to the already overflowing Android root store is not a safe method to deliver transport layer security to an application's communications.

## V. Evaluation

For this implementation, PMD was chosen due to its extensible interface and basis in source code parsing [8]. Error messages will inform developers more than fix their mistakes. This remains in line with our design of a system that points out developer blindspots rather than correcting them silently or prints out a cryptic stacktrace. We chose the most recent build at the time of our design–PMD build 5.2.1. [19], [20]. Our final design demos can be seen in Figure 2 and Figure 3.

In order to determine the effectiveness of `emphaSSL`, we must determine whether the detected violations via our `emphaSSL` are valid detections of HTTPS development errors in the app's source code regardless of who is developing it. Therefore, we plan to build an automated process to test our `emphaSSL` first and manually review the detected violations later. First of all, in our testing of `emphaSSL` on a sizeable and readily-accessible sample, a number of popular open source applications' source code files were tested. Only applications that listed the INTERNET permission and had significant use of Internet connection (HTTP/HTTPS) were selected. In detail, 75 open source Android application projects consisting of 9,346 Java files totaling over 100,000 lines of code were downloaded onto the local testing machine, a 64-bit Debian Wheezy desktop with a 3.8 GHZ Intel Core i5-3570K CPU and 4GB of RAM. These popular applications were listed on either in the F-Droid repository [21] or on Github and then cloned for our experiment. A bash script was written which ran PMD against these apps, outputting the results to log files which can be analyzed later. The logs contain truncated messages to preserve readability. These log files were analyzed and a manual investigation into the offending files was conducted following the experiment to determine whether the detected violations were valid detections of security errors. The results of these tests will be discussed in the next section.

### A. Results and Discussion

Analysis of the logs from our experiment provides evidence to the ability of our `emphaSSL` to detect HTTPS misuse and appropriately notify developers as they work. Furthermore, our experiment finds 30 applications (40%) which can be considered vulnerable, 9 of which (12% of all applications in the sample) have serious HTTPS overriding capabilities. Table II and III list the results from the experiment on 75 applications. Table II displays the number of violations of each rule overall. In this table, Rule 1 and 8 are seperate to differentiate AllowAllHostnameVerifier()(Rule 1) and SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER(Rule 8).

**SR. 1 Use of HTTPS Results**: As evidenced by our results, the most common mistake made by developers of this sample is the use of HTTP over HTTPS. Rule 5 violations, which have detected HTTP URLs that do not resolve to valid server addresses, were found in 64 of the 75 applications (85%). These are not as significant in the scope of error catching due to the high likelihood of false positives (such as in the case of string concatenation) as Rule 6. Moreover, 37 of these applications (57% of previous violation pool, 49% of the total application sample) contain Rule 5 violations, but no Rule 6 violations. In 27 of the 75 works (36%), a Rule 6 violation was found, denoting an instance where a secured HTTPS session could be made between the application and the server if an 's' is added to the URL. In general, Rule 6 violations are considered vulnerable unless otherwise determined during the follow-up analysis.

**SR. 2 Hostname Verification Results**: In reviewing our results, there were 31 hostname verification violations overall. Nine projects (12%) had hostname verification overrides in some capacity. The most common override found was Rule 10, the creation of an SSLSocket instance without hostname verification. When combined, Rule 1 and 8, which involve allowing all hostnames, are also present in these projects. All nine projects, which violate hostname verification, contain one of these two rules. Every instance that custom hostname code was written, the intention of which was to ignore hostname checks.

**SR. 3 Valid Error Checking Results**: There were twelve instances of invalid error checking within our sample, however only six projects (8%) have the Rule 9 violation. Each instance occurs twice in the project, a trend which we will later discuss. All projects with this violation are considered vulnerable due to the aforementioned failure to disclose potential security risks to the end user.

**SR. 4 Proper Trust Management Results**: The security requirement with the fewest violations was trust management. Rule 4, which checks for custom trust managers found only six instances (8%) within the sample. Improper trust management is considered a critical vulnerability due to the dangerous attack surface that a bulky root store provides. Custom trust managers can ignore certificate checks or allow self-signed certificates to authenticate servers or middle-man devices.

Following the automated test of these 75 applications, a manual analysis of the project files was conducted. We used the result logs from our automation script to find errors and determine their severity within the offending codebase. In several instances, such as in the case of the WordPress application, proper measures were taken to handle exceptions, but certain blacklisted methods were used in order to allow functionality. The discussion of our results is informed by this manual analysis. In this section we will explore our findings in comparing the automated and manual evaluations.

In general, there were nine projects which have alarming HTTPS overrides. Six of these projects have comparable patterns of violations. Each of the six applications violates Rule 9, either Rule 1 or 8, and Rule 2. Thus, both error checking and hostname verification are turned off, leaving the applications vulnerable to prevailing man-in-the-middle attacks. The instance of six projects using a similar set of blacklisted methods may be the result of a solution being

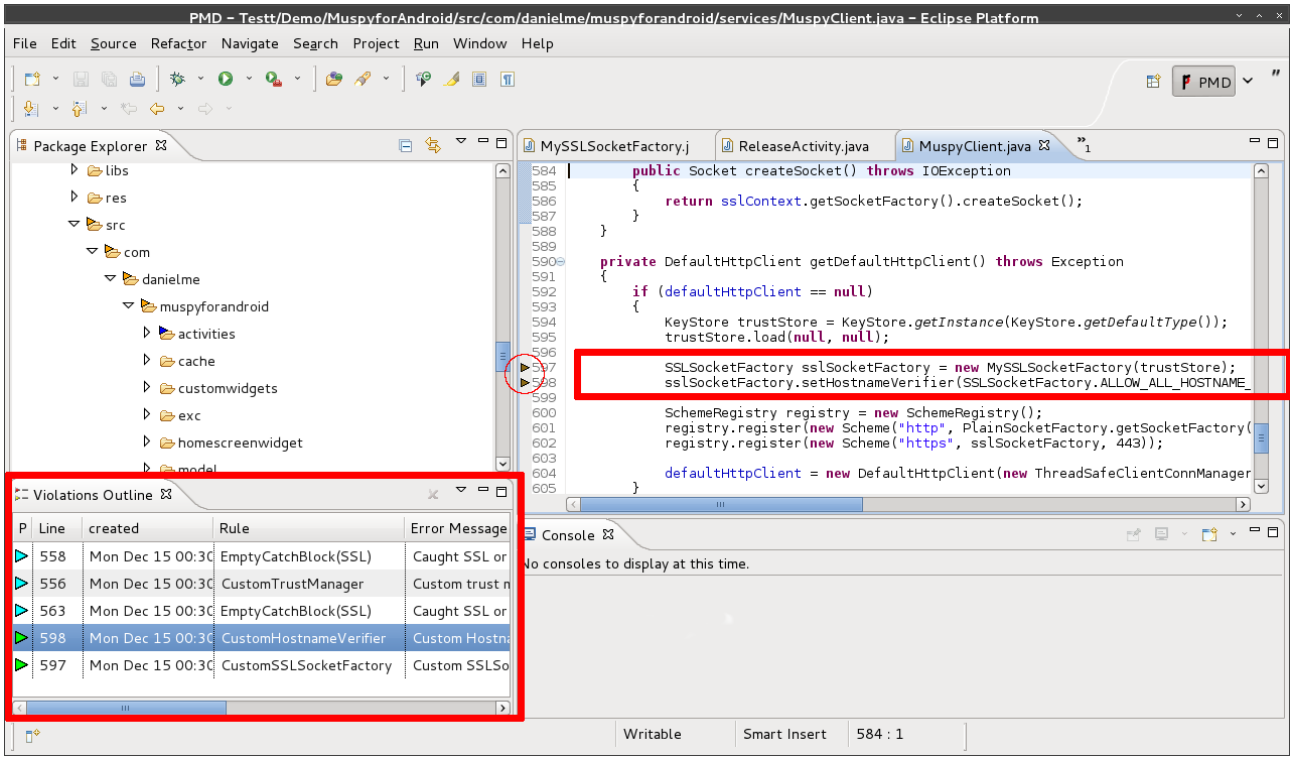


Fig. 2. Displaying PMD error markers and violations outline to the left of violations

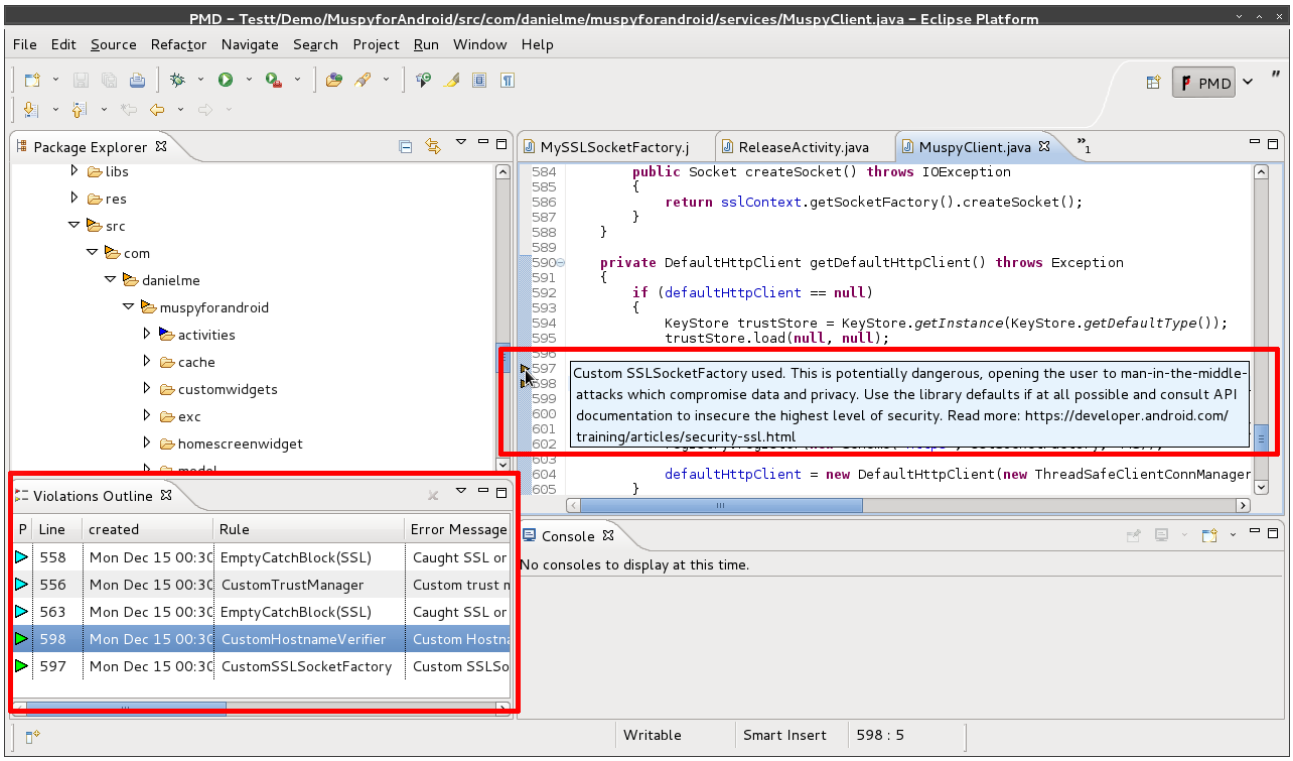


Fig. 3. Displaying a rule message when hovering over a violation tag

extracted from StackOverflow despite the warnings against using the code in production [22].

In a few projects where no violations were noticed, the code which processed the HTTPS was too low-level for the ruleset to detect. Despite the necessity of applications to use generally well-reviewed and trusted implementations of the SSL/TLS protocol, these applications, such as ICS OpenConnect, do not use a library which has methods checked

TABLE II
VIOLATIONS BY RULE

| Rule Name (Rule Number) | Number of Violations |
|---|---|
| AllowAllHostnames (1) | 4 |
| CustomHostnameVerifier (2) | 6 |
| CustomSSLSocketFactory (3) | 3 |
| CustomTrustManager (4) | 6 |
| MoveToHTTPS (5) | 751 |
| HTTPSCouldBeUsed (6) | 115 |
| SSLSetVerifyResult (7) | 0 |
| AllowAllHostnames (8) | 7 |
| EmptyCatchBlock (9) | 6 |
| UnsafeHostnameVerification (10) | 11 |

TABLE III
VIOLATIONS BY PROJECT

| Rule Name (Rule Number) | Number of Violating Projects |
|---|---|
| AllowAllHostnames (1, 8) | 9 |
| CustomHostnameVerifier (2) | 6 |
| CustomSSLSocketFactory (3) | 3 |
| CustomTrustManager (4) | 6 |
| MoveToHTTPS (5) | 37 |
| HTTPSCouldBeUsed (6) | 27 |
| SSLSetVerifyResult (7) | 0 |
| EmptyCatchBlock (9) | 10 |
| UnsafeHostnameVerification (10) | 4 |

by these sample rules. While several of the applications have unique implementations of HTTPS, a few applications have outlier qualities that were identified in the manual analysis. The WordPress application contains a special socket factory that is used when the certificate is not accepted, allowing the user to continue if they click 'OK'. This is actually the correct way to deal with a failed certificate, but was still picked up by our sample ruleset which did not have the context to detect the false positive. The Subsonic Android app accepts self-signed certificates. This is dangerous, but most likely comes from a functional perspective when dealing with home streaming servers. In this case, the security requirements cannot be met while maintaining functionality.

In these instances, the shortsighted nature of a source-code parsing plugin can be seen. Despite this, the warnings brought up by our plugin have identified security violations on a majority of the projects tested without hindering the functionality of the applications. With our `emphaSSL`, developers are able to better identify and prioritize their mistakes, contemplate the implications, and correct them as they see fit.

Following our analysis of the results, we posted code issues on 16 of the 30 'vulnerable' projects which we had determined needed to implement HTTPS. Three of the projects which we issued bug reports on already had notifications of SSL/TLS issues or insecurities. One month later, positive responses to the issue postings were received, which showed actions would be taken in the immediate future to resolve the vulnerability

Our results show that our deisgn is an effective way to spot mistakes in application source-code in a non-invasive fashion. While these the ruleset may lack a contextual depth, they narrow down thousands of lines of source code into a few error markers which a developer can then utilize in hardening Internet security in Android apps.

## VI. CONCLUSION

Our `emphaSSL` has presented a novel method of identifying SSL/TLS vulnerabilities in code which is being written. We also have presented a design for developing rules to prevent HTTPS misuse on the Android platform and a set of ten sample rules. Finally, we have discussed the results of an experiment on seventy-five open source applications showing significant vulnerabilities in 30 (40%) projects. This paper puts forward a more developer-centric integration of security warnings which fits into the paradigm of solving for developer 'blindspots'. Our methodology can positively impact the security of application development before a product reaches compile time. We bridge a gap in the developer's understanding of HTTPS that has been shown in innumerable studies while maintaining the functionality of the program, which ultimately hardens the networking security in Android apps.

## REFERENCES


[1] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *ACM CCS*, 2013.

[2] B. Berger, M. Bunke, and K. Sohr, "An android security case study with bauhaus," in *IEEE WCRE*, 2011.

[3] W. Enck, D. Octeau, P. McDaniel, and S.Chaudhuri, "A study of android application security," in *USENIX Security*, 2011.

[4] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in)security," in *ACM CCS*, 2012.

[5] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *ACM CCS*, 2013.

[6] R. Anubhai, D. Boneh, M. Georgiev, S. Iyengar, S. Jana, and V. Shmatikov, "The most dangerous code in the world: Validating ssl certificates in non-browser software," in *ACM CCS*, 2012.

[7] V. Tendulkar and W. Enck, "An application package configuration approach to mitigating android ssl vulnerabilities," in *MOST*, 2014.

[8] PMD, http://pmd.sourceforge.net/.

[9] P. Chen, M. Dean, D. Ojoko-Adams, H. Osman, L. Lopez, and N. Xie, "System quality requirements engineering (square): Case study on asset management system," Technical Report, Carnegie Mellon University, 2004.

[10] S. Gordon, T. Stehney, N. Wattas, and E. Yu, "System quality requirements engineering (square): Case study on asset management system, phase ii," Technical Report, Carnegie Mellon University, 2005.

[11] C. Haley, J. Moffett, R. Laney, and B. Nuseibeh, "A framework for security requirements engineering," in *ACM SESS*, 2006.

[12] D. Hatebur, M. Heisel, and H. Schmidt, "A pattern system for security requirements engineering," in *IEEE ARES*, 2007.

[13] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *ACM CCS*, 2009.

[14] EFF, "HTTPS Everywhere," 2015, https://www.eff.org/https-everywhere.

[15] Security with HTTPS and SSL, https://developer.android.com/training/articles/security-ssl.html.

[16] J. Cappos, Y. Zhuang, D. Oliveira, M. Rosenthal, and K. Yeh, "Vulnerabilities as blind spots in developers heuristic-based decision-making processes," in *ACM NSPW*, 2014.

[17] "OpenSSL," http://www.openssl.org/.

[18] N. Vallina-Rodriguez, J. Amann, C. Kreibich, N. Weaver, and V. Paxson, "A tangled mess: The android root certificate stores," in *ACM CoNEXT*, 2014.

[19] PMD 5.2.1, http://pmd.sourceforge.net/pmd-5.2.1/.

[20] PMD-Eclipse, http://sourceforge.net/projects/pmd/files/pmd-eclipse/.

[21] F-Droid, https://f-droid.org/.

[22] Stack Overflow: SSL–Untrusted Certificate Error, http://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https.