SSLDoc: Automatically Diagnosing Incorrect SSL API Usages in C Programs

Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Ming Gu
School of Software Engineering, Tsinghua University, Beijing , China, 100084

***Abstract*—Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols provide a reliable communication channel between applications over the Internet. Implementations of these protocols (e.g., OpenSSL and GnuTLS) publish well-format documentation and examples online to guide the usage of SSL/TLS APIs. However, incorrect usages have caused many severe vulnerabilities (e.g., privilege escalation, denial of service, man-in-the-middle attack, etc.) in recent years. In this paper, we introduce SSLDoc to diagnose incorrect SSL API usages in real-world C programs automatically. The key insight behind SSLDoc is a constraint-directed static analysis technique powered by domain-specific usage patterns that we learn from real-world vulnerabilities and bug-fix-related patches. We have instantiated SSLDoc for OpenSSL APIs and applied it to large-scale open-source programs. SSLDoc found 45 previously unknown security-sensitive bugs in OpenSSL implementation and applications in Ubuntu. We created and submitted issues for all of them. Up to now, 35 have been confirmed by the corresponding development communities and 27 have been fixed in master branch.**

***Index Terms*—SSL, API usage validation, static analysis, bug detection**

I. Introduction

Secure Socket Layer (SSL) and Transport Layer Security (TLS) are the most widely deployed protocols in security-sensitive software. They provide a confidential and authentic end-to-end communication mechanism against an active, man-in-the-middle attacker. The details of these protocols are complicated and involves many steps to set up and validate certificate authority [1], [2]. Therefore, client programs usually rely on SSL libraries such as OpenSSL [3] and GnuTLS [4], which encapsulate the internal details and diverse kinds of cryptography algorithms into APIs with well-format documentation and examples. However, correct usage of SSL APIs is required to satisfy certain constraints, such as call conditions or call orders. Violations of these constraints will lead to software bugs and more critically, can have severe security implications. For example, missing error status code validation of SSL APIs will cause a denial of service by remote attackers (CVE-2016-2182 [5]), and broken SSL certificate validation will result in man-in-the-middle attacks [6]. A recent study show that SSL certificate validation is completely broken in many security-critical applications and libraries [7].

Many different tools, techniques and methodologies have been proposed to address the above problems. Clark et al. [8] present a comprehensive survey of SSL issues to enhance the certificate infrastructure used in practice. Brubaker et al. [9] systematically test the correctness of the certificate validation logic in SSL/TLS implementations. However, they focus on SSL implementation and require considerable manual efforts to prepare a test environment.

To automatically detect incorrect usages of SSL APIs in client programs, static analysis has long prevailed as one of the most promising techniques [10]. For example, He et al. [11] design and implement SSLINT, a scalable static analysis tool to match a program dependence graph with a handcrafted, precise signature modeling the correct logic usage of SSL APIs. Although SSLINT is capable of detecting incorrect usages in practice, it is hard to apply to APIs without predefined signatures and produces many false positives and false negatives due to imprecise static analysis (e.g., flow-insensitive and context-insensitive). Yun et al. [12] present APISan for incorrect API usages of causal relation and semantic relation on arguments with security implications by leveraging the strength of static analysis (such as control dependency analysis) and code mining (such as frequent sub-itemsets mining algorithm). It provides accurate detection and can be applied to scale real-world system programs. However, a challenge for such tools is insufficient data to train models, which is particularly severe for SSL APIs in client programs.

In this paper, we aim at augmenting current detection capability of incorrect SSL API usage for large-scale C programs. The key insight is a constraint-directed static analysis technique powered by domain-specific usage patterns. To understand the root causes of incorrect usages of SSL APIs, we begin with a preliminary investigation of real-world vulnerabilities to summarize generic incorrect usage patterns. Leveraging this knowledge, we design and implement SSLDoc, a static analysis detector employing under-constrained symbolic execution [13] to generate abstract symbolic traces with rich semantics and detect incorrect usages. In this way, SSLDoc can precisely conduct a flow-, control- and context-sensitive analysis inter-procedurally (i.e., capable of capturing temporal sequencing of API calls, path constraints, and data flows between parameters and return values in or across procedures).

To evaluate SSLDoc in practice, we instantiated it with OpenSSL APIs and applied it to more than half million lines of source code, including OpenSSL implementation and 15 applications in Ubuntu. The result shows that SSLDoc discovers 45 previously unknown security-sensitive incorrect SSL API usages. We reported our findings to developers and received 35 confirmations, out of which 27 have been fixed in multiple branches. Moreover, we share the lessons

DOI reference number: 10.18293/SEKE2019-006

learned from bug detecting, issue reporting and discussions with developers.

In summary, our paper makes the following contributions:

- We design and implement SSLDoc, a static analysis tool to augment current detection capability of incorrect SSL API usage for large-scale C programs.
- We instantiate SSLDoc with OpenSSL APIs and apply it to real-world programs. It discovers 45 previously unknown incorrect SSL API usages, out of which 35 have been confirmed by developers.
- We share the lessons learned from bug detecting, issue reporting and discussions with developers in practice. We hope our findings can motivate more researcher to combat incorrect SSL API usages.

The rest of this paper is organized as follows. Section II provides motivating examples of our work. Section III presents the design of SSLDoc, followed by an evaluation in Section IV. We share the lessons learned in Section V and discuss related work in Section VI and conclude in Section VII.

II. Motivating Example

Instead of implementing SSL themselves, Client programs usually rely on APIs of SSL libraries such as OpenSSL and GnuTLS as well as higher-level data-transport libraries such as Curl [14]. While APIs encapsulate the details, they also expose rich semantic constraints. Violations of these constraints, in turn, lead to serious security problems.

To better understand incorrect SSL API usage patterns and how developers fix them in practice, we manually studied four years' (from 2013 to 2017) CVE entries related to API usage bugs in National Vulnerability Database[1]. They are extracted through approximate keywords matching (e.g., "OpenSSL API usage" and "incorrect SSL usage") and contain concrete patches to fix the bugs. We investigate both the CVE description messages and patches, and identify two generic incorrect usage patterns as shown in Figure 1:

- **Certificate Validation.** SSL libraries encapsulate the core functionality of protocols and export APIs to utilize the implementation. However, the client needs to validate all kinds of certificates in applications. Missing validations might allow attackers to cause a denial of service or man-in-the-middle via an invalid one. Figure 1a shows an example of such vulnerabilities reported in CVE-2015-0288 [15]. Function `X509_get_pubkey()`[2] attempts to decode the public key for *x*. If an error occurs, it will return NULL. In function `X509_to_X509_REQ()`, the return value *pktmp* is used without checking the error code, which results in a NULL Pointer Dereference bug. Beyond null pointer checking, SSL libraries use various error protocols in practice (e.g., 0 or negative for errors in OpenSSL, but -1 to -403 in GnuTLS).
- **Causal Function Calling** SSL libraries allocate memory resources for cryptography algorithm computing, which should release after their lifecycle by invoking a causal function calling. Violations of such causal relation (i.e., a-b pattern) will cause a denial of service (memory consumption) via an intentionally crafted input by remote attackers. For example, `EVP_CIPHER_CTX_cleanup()`[3] clears all information from a cipher context *ctx* and free up any allocated memory associated with it. However, missing invoking it along the error handling path of `tls_decrypt_ticket()` will be exploited by a crafted session ticket that triggers an integrity-check failure as shown in Figure 1b.

```
1  Location: OpenSSL/crypto/x509/x509_req.c: 70
2  X509_REQ *X509_to_X509_REQ(...){
3    [...]
4    pktmp = X509_get_pubkey(x);
5  // missing certificate validation of pktmp
6  + if (pktmp == NULL)
7  +    goto err;
8    i = X509_REQ_set_pubkey(ret, pktmp);
9    EVP_PKEY_free(pktmp);
10   [...]
11 }
12 ===== Correct Usage =====
13 Location: /crypto/x509/x509_cmp.c: 390
14 int X509_chain_check_suiteb(...){
15   [...]
16   pk = X509_get_pubkey(x);
17   rv = check_suite_b(pk, -1, &tflags);
18   [...]
19 }
20 static int check_suite_b(EVP_PKEY *pkey,...){
21   [...]
22 // ensure pkey not NULL
23   if (pkey && ...)
24      [...]// error handling
25 }
```

(a) Incorrect usage for missing certificate validation reported in CVE-2015-0288 [15].

```
1  Location: OpenSSL:ssl/t1_lib.c: 3567
2  static int tls_decrypt_ticket(...){
3    EVP_CIPHER_CTX ctx;
4    [...]
5    EVP_CIPHER_CTX_init(&ctx);
6    [...] // Check HMAC of encrypted ticket
7    if (CRYPTO_memcmp(tick_hmac, etick + eticklen, mlen))
8  + {  EVP_CIPHER_CTX_cleanup(&ctx);
9       return 2;
10 +}
11   [...]
12   sdec = OPENSSL_malloc(eticklen);
13   if (!sdec){
14      EVP_CIPHER_CTX_cleanup(&ctx);
15      return -1;}
16   [...]
17   EVP_CIPHER_CTX_cleanup(&ctx);
18   [...]
```

(b) Incorrect usage for missing releasing resource reported in CVE-2014-3567 [16].

Fig. 1: Motivating examples of incorrect SSL API usages.

Detection of the above bugs is not trivial. It requires a wide spectrum of semantics instead of simply syntactic matching. For example, function invocation of `X509_get_pubkey()` at Line 16 in Figure 1a is correct, because `check_suite_b()` validates the first parameter to ensure that *pkey* is not NULL. To filter out such instance, it

[1]http://cve.mitre.org/

[2]https://www.openssl.org/docs/manmaster/man3/X509_get_pubkey.html

[3]https://www.openssl.org/docs/man1.0.2/crypto/EVP_CIPHER_CTX_cleanup.html

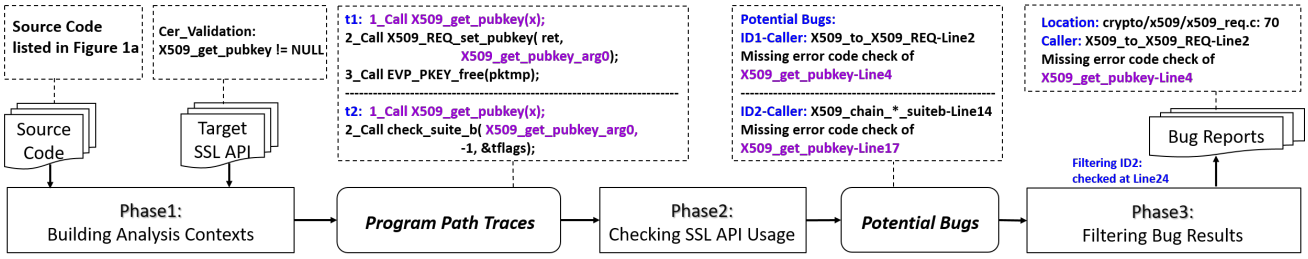

Fig. 2: Overview of SSLDoc's workflow.

demands a flow- and context-sensitive semantic analysis interprocedurally. Moreover, path-sensitive analysis also should be taken into consideration to check memory leak along different error handling paths (e.g., Line 14 and 17 in Figure 1b).

III. Approach

In this section, we introduce SSLDoc, a static analysis tool to augment current SSL API usage detection capability for large-scale C programs. We first present a brief overview of our approach with an example of Figure 1a and elaborate each step of bug detection in the following parts.

As shown in Figure 2, SSLDoc takes the source code and target APIs as input and generates bug reports with concrete locations and reasons as output. Bug detection consists of three basic steps. (1) In **Phase-1**, the analysis context is built by constructing the control flow graph and creating *program path traces* for each target API by employing under-constrained symbolic execution. In this example, two traces, t1 and t2, are generated, as shown in the box above ***Program Path Traces***. In this way, SSLDoc can successfully capture the usage context of `X509_get_pubkey()`, `EVP_PKEY_free()` and those in between. (2) In **Phase-2**, SSLDoc employs the *traces* to detect violations of API usages as potential bugs. For example, two API-misuse instances of `X509_get_pubkey()` are found for missing certificate validations labeled as *Potential Bugs*. (3) In **Phase-3**, SSLDoc improves the detection precision by leveraging inter-procedural semantics and usage statistics. Then, the second misuse is filtered out for the check conducted in the `X509_to_X509_REQ()` at Line-24. We discuss the details of our approach as follow.

A. Building Analysis Contexts

SSLDoc performs symbolic execution to generate program path traces that capture rich semantic information for each target API. In Figure 3, we illustrate the workflow for building analysis context, which consists of three steps. First, SSLDoc parses the source code and builds a control flow graph (CFG). Then, for each target API f, we select analysis entries as target API call sites by labeling the callers C, which invokes f. Next, for each caller $c \in C$, symbolic execution is employed to generate a series of program path traces T with rich semantics of usages of f while traversing the CFG.

We use $\mathbb{N}, \mathbb{Z}$ to denote the set of non-negative and all integers, respectively. In Figure 4, we formally describe the structure of program path traces computed by SSLDoc, where $id \in \mathbb{N}, n \in \mathbb{N}, z \in \mathbb{Z}$ and ap is short for Accesspath [17] to represent memory locations in the form of regular expressions. Each trace t consists of a sequence of actions a^+ with a value map V. In particular, **Assume** action is used to capture path-sensitive semantics. All the actions are labeled while traversing CFG to support flow-sensitive analysis. V records the semantics from a symbolic variable sv to a concrete value cv. A symbolic variable is defined by an action labeled by id and the index n. For example, $id\_f\_\textbf{arg}\_i$ denotes the i^{th} parameter of f called in the id^{th} action. In this way, we can capture the invocation context semantics. We use $f\_\textbf{arg}\_0$ to represent the return value of f and $\textbf{arg}\_0$ for the symbolic variable returned by the caller c of f in **Return** action. Therefore, our program path trace is capable of capturing the flow-, context- and path-sensitive semantics.

In Figure 5, We illustrate three traces of the example code listed in Figure 1a. t_1 and t_3 are original code snippets, and t_2 is with the bug-fix patch. All traces start from the action calling `X_509_get_pubkey()`. Then, t_1 directly passes the return value **1_X509_get_pubkey_arg_0** into `X509_REQ_set_pubkey()` without certificate validation. By contrast, t_2 validates the return value immediately. Even though t_3 passes it into `check_suite_b()` without valida-

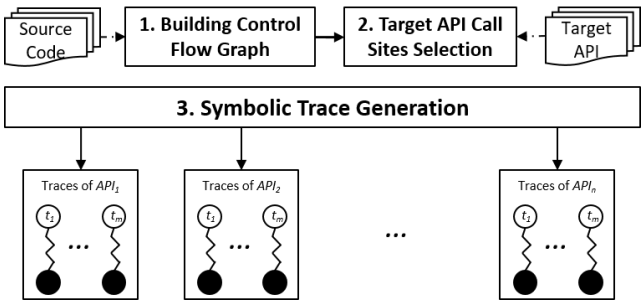

Fig. 3: Workflow of building analysis contexts.

$$
\begin{aligned}
\text{(Traces)}\ & T ::= t \\
\text{(trace)}\ & t ::= (id\_a)^+; V \\
\text{(action)}\ & a ::= \textbf{Assume}(exp) \mid \textbf{Call}\ f(sv^*) \mid \textbf{Return}(sv) \\
\text{(expression)}\ & exp ::= sv1\ \ cmpop\ \ sv2 \\
\text{(value map)}\ & V ::= sv \rightarrow cv \\
\text{(symbolic variable)}\ & sv ::= (id\_f\_\textbf{arg}\_n) \\
\text{(compare operator)}\ & cmpop ::= != \mid == \mid >= \mid > \mid <= \mid < \\
\text{(concrete value)}\ & cv ::= z \mid ap \mid \texttt{NULL} \\
\text{(function)}\ & f \in \mathbb{F}
\end{aligned}
$$

Fig. 4: Abstract syntax of program path traces.

t_1 : 1_**Call** X509_get_pubkey(_);
2_**Call** X509_REQ_set_pubkey(_, **1_X509_get_pubkey_arg_0**) ;
t_2 : 1_**Call** X509_get_pubkey(_);
2_**Assume**(**1_X509_get_pubkey_arg_0 != NULL**) ;
3_**Call** X509_REQ_set_pubkey(_, **1_X509_get_pubkey_arg_0**) ;
t_3 : 1_**Call** X509_get_pubkey(_);
2_**Call** check_suite_b(**1_X509_get_pubkey_arg_0**, _, _) ;
2_**Assume**(**1_X509_get_pubkey_arg_0 != NULL**) ;

Fig. 5: Program path traces of the code in Figure 1a, where we use "_" to represent values irrelevant.

tion, `check_suite_b()` checks the first parameter at Line 24 in Figure 1a.

Similar to the traditional analysis, the key challenge of building such path traces in large and complex programs is to overcome the path-explosion problem. We make two design decisions to achieve scalability without sacrificing substantial accuracy. (1) Limiting inter-procedural analysis. SSLDoc performs symbolic execution intre-procedurally for each caller c of the target API f at most two depth (i.e. we track c and callees of c). We refine the bug detection results by a filtering phase with the deeper inter-procedural semantics presented in Section III-C. (2) Unrolling loops. SSLDoc unrolls each loop only once to reduce the number of paths explored. While this restriction can limit the accuracy of the semantic computation, it does not noticeably affect the accuracy of SSL API bug detection for only a small number of usages related to loop variables.

B. Checking API misuses

In the checking phase, SSLDoc employs target APIs and the program path traces T to detect bugs. To configure usage pattern of SSL APIs, we provide each target f with a usage pattern type $f.\mathcal{T}$, which can be certificate validation with a predicate $\mathcal{P}$ (e.g., X509_get_pubkey != NULL) and causal function calling with a usage pattern $\mathcal{C}$ (e.g., `EVP_CIPHER_CTX_init`(arg1) $\rightarrow$ `EVP_CIPHER_CTX_cleanup`(arg1), where arg1 labels the target memory object). We illustrate our detecting algorithm in Algorithm 1. First, we extract all the target functions into $APISet$. For each API f, we detect API-misuse bugs along the traces T' that invoke f. Then, for each trace t in T', we validate whether the usage pattern in $f.\mathcal{T}$ are satisfied along t. If a path fails, SSLDoc labels the call site of f along this t as a potential bug. To check the satisfaction of certificate validation along t, we compute the satisfiability of $\mathcal{P}$. That is, whether there is an **Assume** action to ensure $\mathcal{P}$. For usage pattern $\mathcal{C}$, we match with **Call** actions, which satisfy the target memory object constraint. If any of the constraints fail to match, SSLDoc reports a bug. For example, t_1 in Figure 5 fails to ensure the certificate validation X509_get_pubkey != NULL, which may result in a null pointer dereference bug.

C. Filtering Bug Reports

To achieve the scalability required to support real-world programs, we employ a limiting inter-procedural strategy to address the path-explosion problem. The strategy generates false positives when a usage cross more than two functions. However, developers dislike using tools with low precision [18]. Therefore, we apply deeper inter-procedural semantics and rank the final results according to usage statistics.

Algorithm 1 Algorithm for checking incorrect SSL API usage

Input: program path traces T, target APIs $\mathbb{F}$
Output: bug report R
1: $R \Leftarrow \emptyset$
2: $APISet \Leftarrow$ extractTargetAPISet($\mathbb{F}$)
3: **for** each API $f \in APISet$ **do**
4: $T' \Leftarrow$ extractPathTraces(f, T)
5: **for** each trace $t \in T'$ **do**
6: $result \Leftarrow$ satisfy($t, f.\mathcal{T}$)
7: **if** ($!result$) **then**
8: $R \Leftarrow$ addBug(t, f)
9: **end if**
10: **end for**
11: **end for**
12: **return** R

First, we conduct semantic-based filtering. We attempt to infer semantics acrossfunctions. For the missing validation of certificate x, we further check the functions which directly receive x as a parameter. If these functions contain sanity check against x, we filter it out. For causal function calling pattern as $a \rightarrow b$, if the target memory object of a is directly assigned to the parameter of the caller c of a or returned by c, we check whether callers $\mathbb{C}$ of c invoke b. We filter out the cases that contain function invocation of b.

Then, we conduct a usage-based ranking. Basically, we compute the number of correct/incorrect usage traces respectively, ranks bug reports in decreasing order of their likelihood of being bugs as $\mathcal{H}(f) = \frac{\#\text{of correct usage traces of } f}{\#\text{of incorrect uage traces of} f}$. The highest likelihood value indicates that more correct usages occur and the violations are less. Therefore, the violations are highly buggy. Note that, we use the trace number instead of call site number, because of the observation that many bugs occur along path-branches with different context semantics. However, we have to specially treat when $\mathcal{H}(f)$ is 0, because it indicates that all the usages are buggy. The preliminary experiment results show that it frequently occurs in small programs which invoke SSL APIs only once or twice.

D. Implementation

SSLDoc is built in Java language. We preprocess the source code into LLVM-IR 3.9[4], which provides a typed, static single assignment (SSA) and well-suited low-level language. Then, we parse the LLVM-IR by javacpp[5] and construct an extended control flow graph, which classifies the edges into control edges for semantic computation and summary edges to provide a mechanism to support large-scale programs. We have integrated part of OpenSSL APIs with SSLDoc and provide an interface to extend our analysis in a human-readable format named Yaml [6].

[4]http://releases.llvm.org/3.9.0/docs/ReleaseNotes.html
[5]https://github.com/bytedeco/javacpp
[6]http://yaml.org/

IV. Evaluation

In this section, we describe our results from incorrect SSL API usage detection on large-scale open-source programs using SSLDoc. We begin by providing the experimental setup. Then we present the security-sensitive bugs we found and concluding with lessons we learned.

a) Experimental Setup: We applied SSLDoc to find incorrect SSL API usages in OpenSSL implementation as well as applications using OpenSSL library in Ubuntu 16.04. Target applications are selected by search dependence attributes using package management command line "`apt-cache rdepends libssl1.0.0`[7]". In total, we found more than 1200 packages using this library and selected 15 packages which are open-source on Github and ongoing development. For all the 16 programs, we detect incorrect usages in the latest stable versions. Then, we use GNU cflow[8] to extract target SSL APIs invoked in the applications and create usage pattern mentioned in Section II according to the user manual of OpenSSL[9]. In total, 136 different SSL APIs are integrated with SSLDoc. We ran SSLDoc on Ubuntu 16.04 LTS (64-bit) with a Core i5- 4590@3.30 GHz Intel processor and 16 GB memory.

b) Result: Overall, SSLDoc detected 45 previously unknown security-sensitive incorrect SSL API usages as listed in Table I. We tried our best to understand the context and created issues for all the bugs to the developers of each program. Up to now, 32 of the new bugs have been confirmed by the developers and 27 have been fixed in the master branch.

For example, in Figure 6 we present a bug caused by incorrect validation of connect status in dma, a small Mail Transport Agent, which is fixed at 12 hours after we submitted the bug report with bug description and explanation of bug traces. Function `SSL_connect()` initiates the SSL handshake with a server. It returns 0 and negative integers to indicate SSL handshake is not successful. However, the status validation in dam/crypto.c only checked against negative integers, which may cause a man-in-the-middle attack leading to leakage of user credentials and emails messages.

V. Discussion

While investigating the bug reports generated by SSLDoc, we find several intricate bugs and gain useful experience in the bug reporting process with open-source developers. We share our following experience. **(1) Incorrect SSL API usages are not corner cases.** In total, we find 45 previously unknown incorrect usages. However, OpenSSL library has provided well-format documentation and examples to guide correct usages. These bugs may result from the lack of a bug information sharing mechanism and the lack of API usage constraints among client software developers. We believe that bug fixing is an essential activity during the entire life cycle of software development. Automatic bug-finding tools,

[7]In Ubuntu16.04 OpenSSL library is listed as libssl1.0.0.
[8]http://www.gnu.org/software/cflow/
[9]https://www.openssl.org/docs/manmaster/man3/

TABLE I: Previously unknown incorrect SSL API usages detected by SSLDoc

Index	Program	Issue ID	Target API	Status
1	OpenSSL 1.1.1-pre8	6567	RAND_bytes	✓✓
2		6568	ASN1_INTEGER_get	✓
3		6569	ASN1_INTEGER_set	✓✓
4		6570	ASN1_object_size	✓
5		6572	BN_set_word	✓✓
6		6573	HMAC_Init_ex	✓
7		6574	EVP_PKEY_get0_DH	✓✓
8		6575	EC_KEY_generate_key	✓
9		6781	EC_GROUP_new_by_curve_name	✓✓
10		6789	ASN1_INTEGER_set	✓✓
11		6820	ASN1_INTEGER_to_BN	✓✓
12		6822	BN_sub	✓✓
13		6973	EVP_MD_CTX_new	✓✓
14		6977	ASN1_INTEGER_set	✓✓
15		6982	OBJ_nid2obj	✓✓
16		6983	BN_sub	✓✓
17		7235	DH_set0_key	✓
18	dma	59	SSL_connect	✓✓
19	exim	2316	X509_NAME_oneline	✓✓
20		2317	SSL_CTX_set_cipher_list	✓✓
21	hexchat	2244	BN_set_word	✓
22		2245	DH_set0_key	P
23	httping	41	SSL_CTX_new	✓
24	ipmitool	37	MD2_Init	✓
25	open-vm-tools	291	SSL_CTX_set_cipher_list	✓✓
26		292	X509_STORE_CTX_get_current_cert	✓✓
27	irssi	943	SSL_get_peer_certificate	P
28		944	BIO_read	P
29	keepalive	1003	SSL_CTX_new	✓✓
30		1004	SSL_new	✓✓
31	thc-ipv6	28	BN_new	✓✓
32		29	BN_set_word	✓✓
33	FreeRADIUS	2309	BIO_new	✓✓
34		2310	i2a_ASN1_OBJECT	✓✓
35	trafficserver	4292	SSL_CTX_new	P
36		4293	SSL_new	P
37		4294	SSL_write	P
38	tinc	205	BN_hex2bn	✓✓
39		306	RAND_load_file	✓✓
40	sslsplit	224	SSL_CTX_use_certificate	✓✓
41		225	SSL_CTX_use_PrivateKey	✓✓
42	rdesktop	280	BN_bin2bn	P
43		281	BN_mod_exp	P
44	proxytunnel	36	SSL_connect	P
45		37	SSL_new	P

✓✓ is fixed, ✓ is confirmed without a patch, and P is wating developr responce.

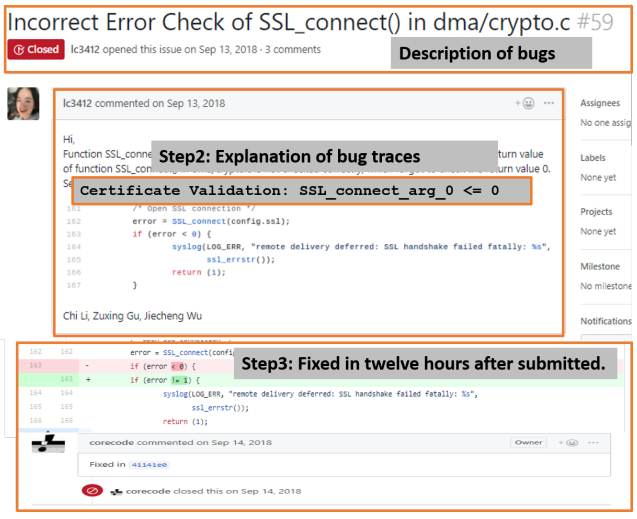

Fig. 6: Screenshot of a bug caused by incorrect validation of SSL_connet() status in dma, which is fixed at 12 hours.

such as SSLDoc, with large-scale analysis capability can be integrated into the development cycle. In addition, SSLDoc can be customized to incrementally address this problem. **(2) Accelerating manual auditing.** SSL API usages usually have similar behavior patterns. For example, many types of vulnerabilities result from insufficient validation of input or missing certificate validations. However, discovering all the missing checks by human is tedious and time-consuming. Automatic tools can efficiently accelerate the manual auditing with differences extracted as good usages and bad usages. For example, two of the API misuses were fixed within 12 hours after we created the issues with possible fixing patches, as shown in Figure 6. **(3) Intentional choices.** We also find that many incorrect usages are not mistakes but intentional choices. Many error status code checks of return values are ignored by developers. During the bug reporting process with the OpenSSL developers, we learned that they intentionally ignore some error code checks for performance considerations or due to the lack of an error handling mechanism in C[10].

VI. Related Work

A few works in the past have analyzed application vulnerabilities due to improper usage of SSL/TLS. Georgiev et al. [7] employ dynamic analysis to conduct MITM attacks and demonstrate that SSL certificate validation is completely broken due to badly designed APIs of SSL implementations. Later, Clark et al. [8] present a comprehensive survey of SSL security and Brubaker et al. [9] apply Frankencerts, a smart fuzzer to test SSL/TLS certificate validation code in implementation. He et al. [11] develop SSLINT, a scalable, automated, static analysis system for detecting incorrect certificate validation vulnerabilities in client programs with predefined API signatures. To automatically infer usage pattern, Yun et al. [12] present APISan to infer correct API usages from source code without manual effort and detect various properties with security implications. Moreover, generic bug detection approaches also can be applied to SSL/TLS API usage, such as static analysis approaches [19], [20] and testing [21]. SSLDoc specifically targets SSL API usages in C programs and complements these works. In addition, our work can be easily extended to other domains.

VII. Conclusion

Client programs rely on APIs of libraries implementing SSL/TLS protocols to ensure reliable communications. Incorrect usage of such APIs will cause security-sensitive problems, even severe vulnerabilities. In this paper, we present SSLDoc, a static analysis detector to automatically diagnose incorrect usages of SSL APIs in C programs. We instantiate SSLDoc with APIs of OpenSSL and apply it to large-scale programs. We find 45 previously unknown bugs in OpenSSL implementation and 15 applications in Ubuntu which use SSL APIs, out of which 27 have been fixed. We share the lessons learned from bug detection and discussions with developers to motivate more researchers and practicers to combat incorrect SSL API usages.

[10]https://github.com/openssl/openssl/issues/6575

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research is sponsored in part by National Natural Science Foundation of China (Grant No. 61802259, 61402248, 61527812), National Science and Technology Major Project of China (Gran No. 2016ZX01038101), and the National Key Research and Development Program of China (Grant No. 2015BAG14B01-02, 2016QY07X1402).

References

[1] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," Tech. Rep., 2008.
[2] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (ssl) protocol version 3.0," Tech. Rep., 2011.
[3] "Openssl: cryptography and ssl/tls toolkit." https://github.com/openssl/openssl, 2019.
[4] "Gnutls: a secure communications library implementing the ssl, tls and dtls protocols and technologies around them." https://gitlab.com/gnutls/gnutls/, 2019.
[5] "Cve-2016-2182," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2182, 2016.
[6] "Cve-2016-2113," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2113, 2016.
[7] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 38–49.
[8] J. Clark and P. C. van Oorschot, "Sok: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements," in *SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 511–525.
[9] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 114–129.
[10] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - test and measurement of static code analyzers," in *COUFLESS 2015, Florence, Italy, May 23, 2015*, 2015, pp. 14–20.
[11] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLINT," in *SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 519–534.
[12] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing API usages through semantic cross-checking," in *USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 363–378.
[13] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 49–64.
[14] "Curl: A command line tool and library for transferring data with url syntax." https://github.com/curl/curl, 2019.
[15] "Cve-2015-0288," https://www.cvedetails.com/cve/CVE-2015-0288/, 2015.
[16] "Cve-2014-3567," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3567, 2015.
[17] B. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," in *PLDI 2000, Vancouver, Britith Columbia, Canada, June 18-21, 2000*, 2000, pp. 57–69.
[18] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[19] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," in *SYNASC 2017*, 2017, pp. 161–168.
[20] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, pp. 1–1 (Early Access), 2018.
[21] M. Kassab, J. F. DeFranco, and P. A. Laplante, "Software testing: The state of the practice," *IEEE Software*, vol. 34, pp. 46–52, 2017.