Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed*

Balázs Kiss[1], Nikolai Kosmatov[2], Dillon Pariente[3], and Armand Puccetti[2]

[1] Search Lab, 1117 Budapest Hungary
firstname.lastname@search-lab.hu
[2] CEA, LIST, Software Reliability and Security Laboratory, PC 174
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[3] Dassault Aviation, 92552 Saint-Cloud France
firstname.lastname@dassault-aviation.com

Abstract. Security of modern information and communication systems has become a major concern. This tool paper presents FLINDER-SCA, an original combined tool for vulnerability detection, implemented on top of FRAMA-C, a platform for collaborative verification of C programs, and Search Lab's FLINDER testing tool. FLINDER-SCA includes three steps. First, abstract interpretation and taint analysis are used to detect potential vulnerabilities (*alarms*), then program slicing is applied to reduce the initial program, and finally a testing step tries to confirm detected alarms by fuzzing on the reduced program. We describe the proposed approach and the tool, illustrate its application for the recent OpenSSL/HeartBeat Heartbleed vulnerability, and discuss the benefits and industrial application perspectives of the proposed verification approach.

Keywords: vulnerability detection, static analysis, program slicing, fuzzing, Frama-C, Flinder, Heartbleed

1 Introduction

The recent Heartbleed bug [6] illustrated once again that critical security flaws can remain undetected by a static or a dynamic analysis technique alone [8]. This paper presents FLINDER-SCA, a novel verification tool for vulnerability detection using a *combination* of static and dynamic analyses, as well as a case study illustrating the capabilities of the proposed combined verification approach to detect recent vulnerabilities at the source code level with reasonable amounts of efforts and computing time. This work has been realized in the context of the STANCE project.

The STANCE project[4] belongs to the European FP7 Research Program and proposes to design and implement validation and verification (V&V) tools to ensure security of industrial software in C, C++ or Java. STANCE builds on the FRAMA-C [7], FLINDER [10] and VERIFAST[5] toolkits and extends their capabilities to handle the

* This work has been partially funded by the EU FP7 project STANCE (grant 317753).

[4] See http://www.stance-project.eu/.

[5] See http://people.cs.kuleuven.be/bart.jacobs/verifast.

aforementioned programming languages and perform security analyses. STANCE studies security properties of industrial applications provided by partners. These are related to an Aeronautic use case (from Dassault Aviation, France), Trusted Computing platforms for embedded systems based on the TPM[6] (from Infineon AG, Germany and TU Graz, Austria), and authentication software for complex distributed networks (from Thales COM, France). The vulnerabilities addressed by STANCE have been classified[4] by using the CWE classification [1] and keeping those vulnerabilities that 1) can be detected in the source code, 2) are written in C, C++ or Java, and 3) are related to the considered application categories.

The original contributions of the present work include

- a new combined verification technique for detection of security vulnerabilities,
- its implementation, FLINDER-SCA, realized in the context of the STANCE project,
- an illustration of its application to the recent Heartbleed vulnerability, and
- a discussion of benefits and application perspectives of the proposed approach.

This paper is structured as follows. Section 2 describes the Heartbleed vulnerability. Section 3 provides on overview of the FLINDER-SCA tool and the associated methodology. Sections 4, 5 and 6 describe the tool components and illustrate them on the case study. Section 7 provides a short tool demo. Section 8 discusses the difficulties of detecting the Heartbleed vulnerability. Finally, Section 9 concludes with the benefits of the approach and some future work.

2 The Heartbleed Vulnerability

The Heartbleed bug [6] was discovered in 2014 in OpenSSL[7], the famous cryptographic library widely used to encrypt communications over the Internet. This bug was identified in the HeartBeat functionality, originally intended to check whether a given server is still alive and able to encipher TCP/IP packets with SSL techniques. How HeartBeat operates is straightforward: a client sends a "keep-alive" message containing a *payload* (a random array of bytes intended to be repeated) as well as the payload's size. In turn, if alive, the server is expected to send the very same payload back to the client. This ensures that the server is — at least — able to copy a message previously received and to forward it back to the sender.

The security issue comes from the fact that the size of the payload is specified by the client, and this size is not checked by the server against the effective payload length — causing it to read past the end of the memory area allocated to hold the payload, which is a typical buffer over-read vulnerability [1]. For instance, if the client sends a 3-byte message and indicates *0xFFFF*(=65535) as the fake size, the server will send the following non-padding data back to the client: the message header and length (1+2 bytes), the 3-byte message itself, then 65532 bytes from the server's heap memory immediately following the payload at the time of processing. Since the memory area allocated to the payload changes with each request, an attacker can repeatedly send such

[6] See http://www.trustedcomputinggroup.org
[7] See https://www.openssl.org.

```
 1 buffer = CRYPTO_malloc(1 + 2 + payload + padding);
 2                                          /* normally payload=16, padding=18 */
 3 bp = buffer;
 4             /* Write response type, payload length and contents into buffer */
 5 *bp++ = TLS1_HB_RESPONSE;          /* store 1-byte response header in buffer */
 6 s2n(payload, bp);                   /* store 2-byte payload length in buffer */
 7 memcpy(bp, pl, payload);            /* copy the payload contents into buffer */
 8 bp += payload;
 9                /* Create random padding to protect against traffic analysis */
10 RAND_pseudo_bytes(bp, padding);
11 r = ssl_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Fig. 1. Extract from OpenSSL/HeartBeat source code (`tls1_process_heartbeat` function)

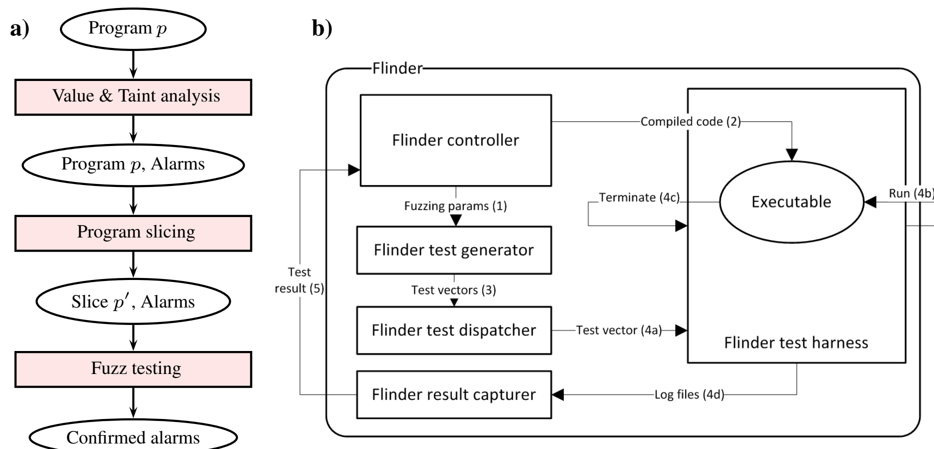

Fig. 2. a) Overview of the proposed methodology, and **b)** Architecture of the FLINDER tool

a request and obtain data stored in many different areas of the heap. Unfortunately, such data may contain confidential data from other processes (e.g. Apache credentials) as well as any other compromising information, and most importantly the secret keys used by OpenSSL itself — which could then be used to impersonate or steal information from the server. Many commonly-used and important Internet sites and their services (such as Google, Youtube, Wikipedia, and Reuters) were compromised by this vulnerability.

The code snippet in Fig. 1, extracted from the OpenSSL/HeartBeat extension v1.0.1+, illustrates the Heartbleed bug. The buffer over-read vulnerability clearly stands in the `memcpy` call statement (line 7). The payload length variable `payload` is indeed specified by the client, possibly an attacker, and determines how many bytes from the payload `pl` will be copied into the buffer starting from buffer pointer `bp`. A few statements later, after adding additional padding bytes (line 10), the contents of the `buffer` variable are sent back to the client (line 11), potentially with a substantial part of the heap.

3 Overview of the FLINDER-SCA Tool

The FLINDER-SCA tool has been realized in the context of two V&V tools: the FRAMA-C code analysis platform [7] , and the FLINDER security testing platform [10]. FRAMA-C provides a collection of scalable and interoperable tools for static and dynamic analyses of ISO C99 source code. It is based on a common kernel that hosts analyzers as collaborating plug-ins that share a common formal specification language. FRAMA-C includes plug-ins based on abstract interpretation, deductive verification and dynamic analysis, as well as a series of derived plug-ins which build elaborate analyses upon the former. In addition, the extensibility of the overall platform, and its open-source licensing, have fostered the development of an ecosystem of independent third-party plug-ins.

The proposed verification methodology is illustrated in Fig. 2a. First, a static analysis step relying on value and taint analyses (detailed in Section 4) is applied to detect *alarms* reporting potential vulnerabilities. Second, a program slicing step (described in Section 5) is used to reduce the initial program p and to produce a smaller one, p', called a *slice*. These two steps are realized by FRAMA-C plug-ins. Finally, the fuzz testing step (presented in Section 6) applies FLINDER on p' to confirm these alarms as actual vulnerabilities. This methodology enhances the SANTE approach [3] that combined value analysis, slicing and structural testing for detection of runtime errors, and makes it well-adapted for detection of security flaws. (For more related work, see [3]).

4 Detection of Alarms by Static Analysis

With FRAMA-C, potential runtime errors (*alarms*) can be detected and localized by the VALUE plug-in [7]. It implements an abstract interpretation based value analysis that computes (over-approximated) domains of possible values for program variables at each program location. For the `memcpy` call responsible for Heartbleed (cf. Fig. 1), VALUE generates the following assertions (slightly rewritten here for the sake of clarity):

```
//@ assert Alarm1: mem_access: \valid(bp[0 .. (payload-1)]);
//@ assert Alarm2: mem_access: \valid(pl[0 .. (payload-1)]);
```

These alarms indicate that the tool cannot ensure the validity of pointers `bp` and `pl` in the range of the payload size `payload`, and therefore dereferencing them may be dangerous.

As VALUE is a sound analyzer [7], it guarantees to generate alarms for all potential runtime errors. It may also generate spurious cases — false positives —, due to over-approximations, especially when users do not provide it with a sufficiently accurate initial state for the inputs. As a result, the alarms of interest with regard to Heartbleed might be raised among numerous other alarms, with no means — at first glance — to *distinguish* preeminent assertions. Of course, more precise analyses could be performed through additional efforts, for instance on the specification of the initial state, or additional annotations in the code to reduce non-conclusive over-approximations. These two workarounds imply a deeper understanding of the application under analysis, and may not be affordable in terms of required efforts or functional expertise in practice.

In this work, we use another approach based on taint analysis [5] to identify code variables and statements concerned with the propagation of *taintable,* i.e. potentially

corrupted inputs. Taintable inputs may contain information controlled by an attacker, and therefore represent a high risk to introduce malicious behaviors. *Taint analysis* allows the user to distinguish which source code statements are concerned with the taintable input flow and are used by a potentially vulnerable function. Taintable data flows are propagated, for instance, in case of pointer aliasing, or copy of memory zones. The proposed taint analysis approach is based on static analysis results computed by VALUE. We have implemented it in an experimental FRAMA-C plug-in.

To apply it on the Heartbleed case, the user specifies the potentially taintable inputs (`rrec.data`, the major part of the HeartBeat message sent by the client), and the vulnerable functions (e.g. `libc` functions `memcpy`, `strcpy`, `fgets`,... that give rise to a significant number of vulnerabilities [4]). The tool reports that the assertions related to `memcpy` call handle the taintable input flow, and the `memcpy` statement is identified as vulnerable[8]. This permits to *distinguish security-related alarms* among all alarms generated by VALUE.

5 Simplification of the Program by Slicing

Program slicing [12, 11] consists in computing the set of program instructions, called *program slice*, that may influence the program state at some point of interest, called *slicing criterion*. Slicing preserves the behaviors of the initial program at the selected criterion after removing irrelevant instructions. It relies on dependency analysis, that can in turn use the results of value analysis.

The SLICING plug-in [7] of FRAMA-C offers various ways to define slicing criteria, including program statements, function calls and returns, read and write accesses to selected variables, and logical annotations. SLICING is also able to handle a conjunction of atomic criteria: by construction, the slice will verify all criteria simultaneously.

In this work, we apply SLICING to simplify the code with respect to the set of alarms produced by static analysis (cf. Section 4). For the program with the Heartbleed vulnerability, initially containing 8 defined functions and 51 lines of code, using SLICING allows us to simplify the code and to keep only 2 defined functions and 38 lines in the slice used in the last step.

6 Confirmation of Alarms by Fuzz Testing

Fuzz testing consists of injecting faulty, erroneous or malformed input into a system under test, and monitoring the state of the system. Detecting an observable error state (such as a crash) indicates that the system cannot properly handle the input in question, confirming the existence of a bug in the code. To be more efficient, fuzzing must be able to generate *syntactically correct*, but *semantically invalid* input by modifying some (sets of) fields within it. The FLINDER fuzz testing framework [10] was originally developed to perform "smart", syntax-aware *black-box fuzzing*: the tester specified the exact format of the input being tested, provided a valid input sample, and defined which of the fields within the format should be modified.

[8] For convenience of the reader, taint analysis results are illustrated in Sec. 7.1.

Within STANCE, FLINDER plays a different role: it is used to determine whether a certain alarm identified by static analysis is an actual vulnerability. FLINDER accomplishes this via *white-box fuzzing:* a specific function inside a program becomes the system under test, and its parameters define individual input fields to be modified. The main white-box operation steps, labelled **(1)**–**(5)**, are shown in Fig. 2b:

(1) Based on the previously-instrumented code (with the potentially vulnerable callsites detected by e.g. value analysis) and information about the particular variables to modify in a function (provided by e.g. taint analysis), FLINDER generates a list of fuzzing parameters for each variable to be modified, specifying what kind of values should be generated for them to look for certain kinds of vulnerabilities.

(2) The instrumented code is compiled and fed to the FLINDER test harness.

(3) Test vectors are generated according to the fuzzing parameters — e.g. strings of varying length for a string variable to identify buffer overflow problems, and very small and very large values for an integer variable to identify integer overflow and array overindexing issues.

(4) Each test vector is sent to the test harness **(4a)**, where its values are used to replace the values in the variables targeted by the fuzzing at runtime **(4b)**. The test harness observes the termination of the function **(4c)**, detects anomalies thanks to the instrumentation, and logs the results **(4d)**.

(5) Based on the presence of anomalies in the logs — such as invalid memory accesses or crashes — FLINDER decides whether the vulnerability is confirmed or not.

In the Heartbleed example, the static analysis step reports to FLINDER six potential bugs, while the slicing step reduces the code and the number of parameters of the function `tls1_process_heartbeat`. Next, the code is instrumented to be able to detect memory violation errors. Used in the white-box mode, FLINDER generates test cases for modifying each of the parameters in turn: 10 test cases for a different-size Heartbeat message buffer, and 32 test cases each for different Heartbeat message length and sequence number values. The first test case where the Heartbeat message length is larger than the buffer size causes an invalid memory read attempt. Captured by the test harness, this operation allows FLINDER to identify the specific FRAMA-C alarm connected to the test. FLINDER ultimately relays this information to FRAMA-C, which can then change the status of the corresponding alarms to *confirmed* (showing them in red in the FRAMA-C GUI)[9].

7 Tool Demonstration

7.1 Static Analysis Step Applied to the Heartbleed Vulnerability

Fig. 3 provides a screenshot illustrating how the first step of FLINDER-SCA allows the verification engineer to detect potential vulnerabilities within the FRAMA-C toolset. The culprit `memcpy` statement is identified as vulnerable, because it manipulates a taintable data flow. We extended the original FRAMA-C GUI by some complementary columns to ease the localization of vulnerable statements in the source code. In the upper left

[9] For convenience of the reader, fuzzing results are illustrated in Sec. 7.2.

panel, several columns identify functions comprising taintable data flows, vulnerable statements and alarms. The upper right panel shows the source code with the taintable data flows and vulnerable statements highlighted in orange and pink respectively. This provides the verification team with a user-friendly overview of taint analysis results on the code under review (especially thanks to the causality with taintable input parameters).

7.2 Fuzz Testing Step Applied to the Heartbleed Vulnerability

In this example, FLINDER is applied to the simplified version of the Heartbleed vulnerability (see Fig. 1). The static analysis step has identified six potential bugs in the `tls1_process_heartbeat` function, and the slicing step has simplified the program to reduce the size and complexity of the code. After appropriately instrumenting the sliced code at each alarm location where memory issues are suspected, FLINDER determines which fuzzing rules to apply — in this case, simple integer fuzzing is applied to the two integer parameters of the function `tls1_process_heartbeat`, and binary data fuzzing is applied to the string parameter (see Fig. 4). Fuzzing the first (string) parameter `s_s3_rrec_data` proves to be inconclusive: injecting modified values into the program does not result in crashes or other incorrect operation. Regardless, fuzzing buffers such as this is important — in many cases, they can contain important data that can affect the execution path of the application. Changing the second (integer) parameter `s_s3_rrec_length` to a value that is larger than the size of the buffer results in an invalid memory access, which is then detected by the test harness due to the hooks inserted into the code. This allows FLINDER to confirm the presence of the vulnerability. Finally, this information is sent back to FRAMA-C to set the status of the corresponding alarms as *confirmed* (in other words, the corresponding assertions are marked in red as invalid, see Fig. 5).

8 Discussion

According to [8], the main difficulties in detecting Heartbleed with static analysis tools were four-fold: the way data is stored and referenced, complexity of following the execution path, difficulty of identifying the specific parts in the storage structure that are misused, and resistance to taint analysis heuristics due to the difficulty of determining whether a specific part within a complex storage structure has become untainted.

Detecting the bug via dynamic analysis ran into another problem: the custom memory management used by OpenSSL would prevent dynamic testing frameworks such as VALGRIND [9] from being able to successfully detect a memory corruption or over-read problem. This — combined with encapsulation of the heartbeat length field within the payload — made its detection via fuzz testing infeasible.

In the end, Heartbleed was detected with two main approaches: Neel Mehta (Google) found it using manual code review[10], and Codenomicon found it through the use of a

[10] as reported by Andrew Hintz, Google vulnerability analyst, see https://news.ycombinator.com/item?id=7558015.

hybrid fuzzer / dynamic analyzer tool. The latter approach is very interesting from a tool standpoint: instead of relying purely on fuzzing, an additional mechanism was employed to detect when the output of a system was semantically incorrect in several ways (bypassing authentication, data leakage, amplification, and weak encryption) [2]. This approach requires additional manual work in the creation of additional information to describe the output, but this only needs to be done once for each interface.

This trend of combining fuzz testing tools with other static and dynamic analysis techniques proves to be an important way of detecting complex and non-obvious security vulnerabilities, moving forward.

To summarize, complex vulnerabilities such as Heartbleed present significant challenges to state-of-the-art static and dynamic analysis tools. While manual code review can always be effective, it is not always a viable solution due to the sheer volume of source code to be inspected in some cases. Thus, new approaches — such as the one proposed in the present work — that combine existing methods are essential in their capacity to detect vulnerabilities automatically without requiring significant manual effort.

9 Conclusion and Future Work

The difficulties of detecting the Heartbleed vulnerability by a static or a dynamic analysis technique alone have been identified and discussed in [8]. To address such vulnerabilities, this work proposes an innovative *combined* approach whose different steps are *complementary* and offer a very promising *synergy.* First, value analysis reports potential errors as alarms, while taint analysis identifies a subset of alarms that are most likely to lead to attacks. Notice that static analysis alone reports several alarms and cannot precisely find the security flaw. Second, slicing reduces the source code by removing statements that are irrelevant w.r.t. the identified subset of alarms. In this case study, slicing reduced the program by 25%, while in earlier experiments on runtime error detection with SANTE [3], the average rate of program reduction by slicing was about 32%. These two steps help to focus on security-relevant alarms in the last step and avoid wasting time by analyzing safe or irrelevant statements. Finally, a fuzz testing step is applied on the reduced code in order to try to confirm the selected alarms. In the present case study, fuzz testing with FLINDER without a preliminary static analysis step could be applied only in a black-box manner and would not be able to find the Heartbleed bug either. Similarly, only using static analysis techniques could not confirm the validity of any identified alarms. Another important benefit for industrial applications of the method is its capacity to detect bugs with *reasonable efforts,* e.g. without the tester having to provide a detailed specification of the input state or additional annotations in the code.

We implemented this method in the Flinder-SCA tool, aiming to connect several new plug-ins developed on top of the Frama-C platform: a taint analysis tool, and a fuzz testing prototype currently being developed within the STANCE project. The originality of the present work with respect to SANTE [3] lies in using taint analysis for identifying the most security-relevant alarms, and fuzz testing for efficient detection

of vulnerabilities. That enhances the SANTE method, adapts it to detection of security flaws and makes it effective for such subtle vulnerabilities as Heartbleed.

FLINDER-SCA is currently used to analyze other proprietary or open-source pieces of software, with negligible adaptations; however, it is important to note that much of the intended vulnerability detection functionality of FLINDER-SCA is still under active development within the STANCE project. Several improvements are planned to enlarge the scope of applications. This concerns in particular the FLINDER tool to address more types of vulnerabilities, and a better integration with taint analysis to be able to apply fuzzing techniques to any control point in the potentially vulnerable workflow under analysis and better identify which parts of the code are the best candidates for fuzzing.

Future improvements also include the investigation of complex input that cannot be represented by variable types — such as a string variable containing an entire SSL3 record consisting of several distinct pieces of data. This can be achieved by adapting Flinder's already existing structure-aware fuzz testing capabilities and employing static analysis methods to help users create the inner structure for such variables as necessary — or in some cases, generating it automatically.

These future developments will permit to apply the methods and tools discussed in this paper to several application candidates, sub-parts of the STANCE project use cases. They could range from basic Apache resource libraries, for which the feasibility can be considered as acquired, to more sophisticated functions (possibly from SingleSignOn software for instance). It is also expected to expand these applications to critical infrastructures in future projects, coupling dynamic and static approaches, in which fuzzing will remain one of the key techniques for verification of complex security properties in complement to classical static analysis methods.

Acknowledgment. We thank the FRAMA-C team for providing the tools and support, and the anonymous referees for many helpful comments.

References

1. CWE-126: Buffer Over-read. http://cwe.mitre.org/data/definitions/126.html
2. Carvalho, M., DeMott, J., Ford, R., Wheeler, D.A.: Heartbleed 101. IEEE Security & Privacy 12(4), 63–67 (2014)
3. Chebaro, O., Cuoq, P., Kosmatov, N., Marre, B., Pacalet, A., Williams, N., Yakobowski, B.: Behind the scenes in SANTE: a combination of static and dynamic analyses. Autom. Softw. Eng. 21(1), 107–143 (2014)
4. Common Vulnerabilities and Exposures. https://cve.mitre.org
5. Denning, D.E.: A lattice model for secure information flow. In: Comm. of the ACM (1976)
6. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
7. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015)
8. Kupsch, J.A., Miller, B.P.: Why do software assurance tools have problems finding bugs like Heartbleed? Continuous Software Assurance Marketplace (Apr 2014)
9. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI 2007
10. Search Lab: Flinder security testing platform. http://www.flinder.hu
11. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)
12. Weiser, M.: Program slicing. In: ICSE 1981. pp. 439–449 (1981)

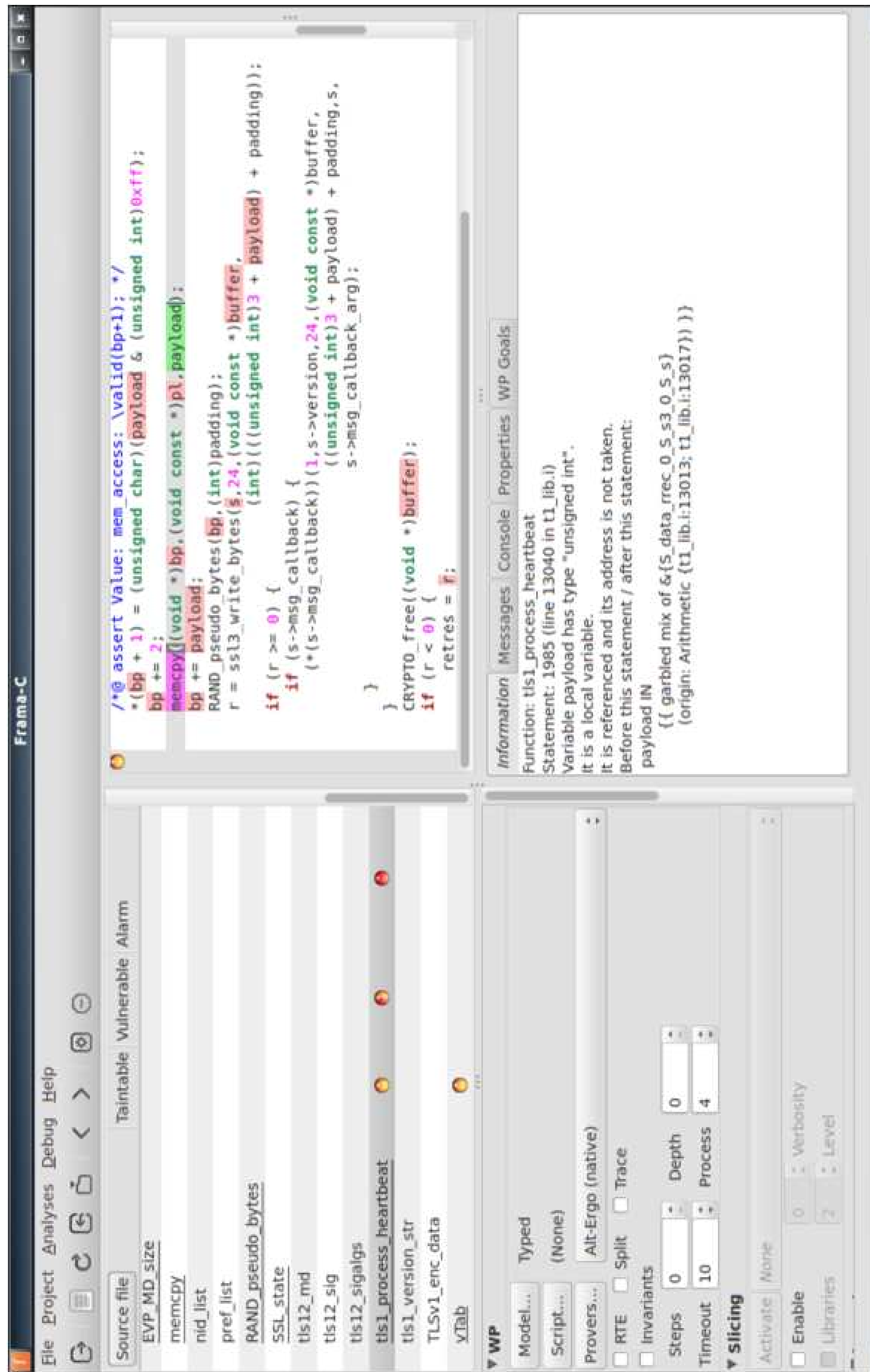

Fig. 3. Frama-C GUI after applying VALUE and taint analysis.

```
Terminal - stance@stance-VirtualBox: ~/Desktop/Examples/FlinderSCA
File  Edit  View  Terminal  Tabs  Help
Test function: tls1_process_heartbeat
Function signature: [{'type': 'unsigned char*', 'name': 's_s3_rrec_data'}, {'type': 'unsigned int', '
name': 's_s3_rrec_length'}, {'type': 'unsigned int', 'name': 's_tlsext_hb_seq'}]
c_to_ast unsigned int
c_to_ast unsigned int
unsigned char
[<convert.c_char_Array_38 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171252792 171252830 39
[<convert.c_char_Array_40 object at 0xb6b134f4>, c_ulong(1), c_ulong(1)]
171324840 171324880 41
[<convert.c_char_Array_44 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171252792 171252836 45
[<convert.c_char_Array_52 object at 0xb6b134f4>, c_ulong(1), c_ulong(1)]
171399656 171399708 53
[<convert.c_char_Array_68 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171399032 171399100 69
[<convert.c_char_Array_100 object at 0xb6b134f4>, c_ulong(1), c_ulong(1)]
171398752 171398852 101
[<convert.c_char_Array_164 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171399184 171399348 165
[<convert.c_char_Array_292 object at 0xb6b134f4>, c_ulong(1), c_ulong(1)]
171398752 171399044 293
[<convert.c_char_Array_548 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171404952 171405500 549
[<convert.c_char_Array_1060 object at 0xb6b134f4>, c_ulong(1), c_ulong(1)]
171402400 171403460 1061
[<convert.c_char_Array_37 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171353440 171353477 38
[<convert.c_char_Array_37 object at 0xb6b134f4>, c_ulong(2), c_ulong(1)]
171353776 171353813 38
[<convert.c_char_Array_37 object at 0xb6b1341c>, c_ulong(1), c_ulong(1)]
171360424 171360461 38
[<convert.c_char_Array_37 object at 0xb6b134f4>, c_ulong(2), c_ulong(1)]
171353776 171353813 38
[<convert.c_char_Array_37 object at 0xb6b1341c>, c_ulong(4), c_ulong(1)]
171360424 171360461 38
[<convert.c_char_Array_37 object at 0xb6b134f4>, c_ulong(8), c_ulong(1)]
171353776 171353813 38
[<convert.c_char_Array_37 object at 0xb6b1341c>, c_ulong(16), c_ulong(1)]
171360424 171360461 38
[<convert.c_char_Array_37 object at 0xb6b134f4>, c_ulong(32), c_ulong(1)]
171353776 171353813 38
[<convert.c_char_Array_37 object at 0xb6b1341c>, c_ulong(64), c_ulong(1)]
171360424 171360461 38
Invalid memory read 171360462
Connection to testharness timed out
Sca test result:  fail
sante_example
set_verdicts called
Number of alarms 5
2
```

Fig. 4. The results produced by FLINDER after applying it to the Heartbleed vulnerability.

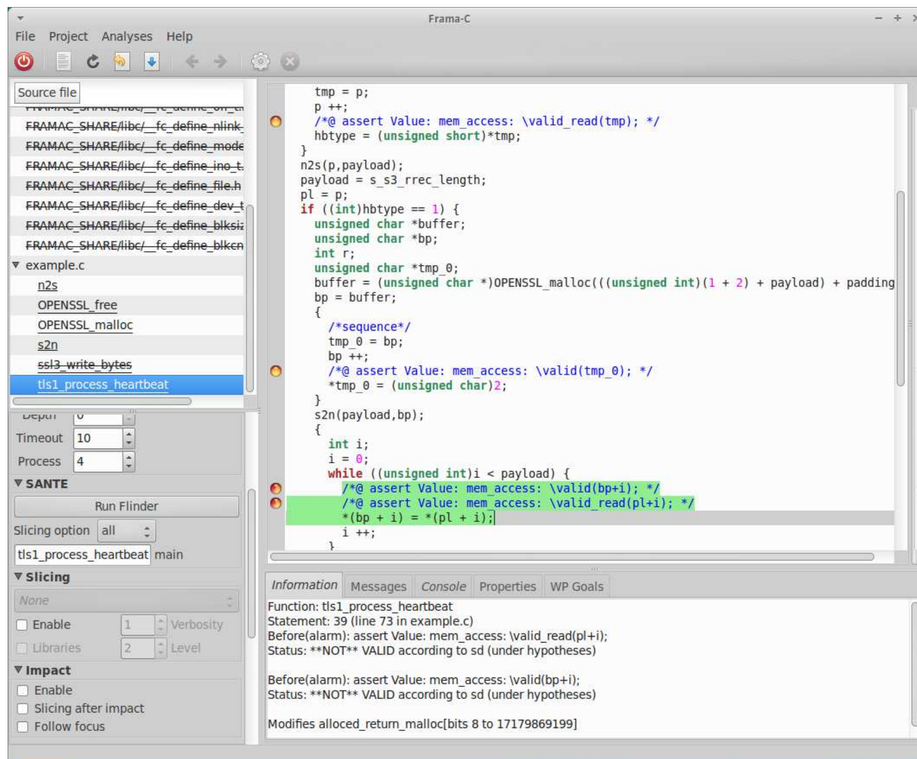

Fig. 5. The final results in the FRAMA-C GUI after applying FLINDER-SCA to the Heartbleed vulnerability. The last two alarms shown in red are real flaws.