Towards Vulnerable Binary Code Clone Detection Through Firmware Lineage

Harrison Fernandez, Computer Science and Information Security Major, John Jay College
Sven Dietrich, Department of Mathematics and Computer Science, John Jay College

Abstract

This poster focuses on code cloning, the reuse of code in same or different programs. Given code cloning tends to create bugs and vulnerabilities, we perform a longitudinal study on the similarity and vulnerabilities in the open source firmware project, Tomato, encompassing more than twelve years of development. This poster explores how clones exist in Tomato, examining the vulnerable binary code of the Busybox tool.

Introduction

Code can be referred to as source or binary.

```
include<iostream>
Using namespace std;
Int main()
{ cout<<''Hello World!''<<endl
   return 0; }
```

```
00000942: 00000010 00000000 01101000 01100101 01101100 01101100 ..hell
00000948: 01101111 00100000 01110111 01101111 01110010 01101100 o worl
0000094e: 01100100 00000000 00000001 00011011 0000011 00111011 d....;
00000954: 01000100 00000000 00000000 00000000 0000111 0000000 D.....
0000095a: 00000000 00000000 01110000 11111101 11111111 11111111 ..p...
```

Figure 1: Source (left) and binary (right) to display “Hello World”.

With the availability of code on the internet (i.e. Github), code cloning is common practice.

- Code cloning affects maintenance and introduces vulnerabilities to large programs [1].
- Vendors of embedded devices are notorious for poor security practices [2], yet routers handle internet traffic, passwords, and more.
- Vulnerabilities may live in software for more than three months without updates or disclosure [3].
- With no way to hold developers accountable, code cloning may affect computer systems and real-time devices.

Research Question

How have vulnerable code clones lived and persisted through software over time?

- Once vulnerabilities are found, are they patched in a proper and timely manner?

Methods

In this study, we analyze the binary and source code of the open source Tomato firmware project.

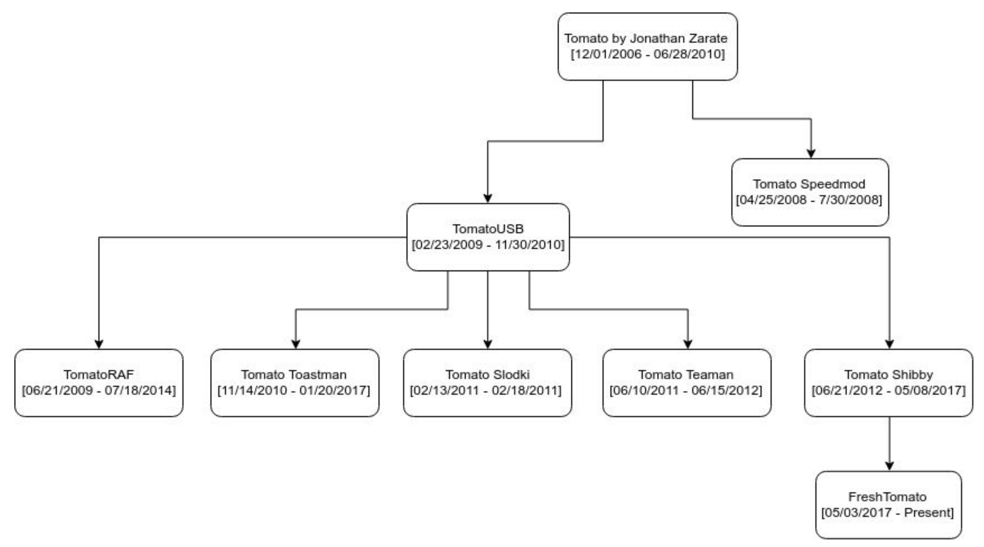


Figure 2: Flow chart of Tomato firmware lineage

- The reverse engineering tool, Radare2, is used to analyze and compare binary files.
- Similarity score is given by Levenshtein distance.
- Note binary similarity between forks may differ due to compiler options and computer architecture, affecting similarity scores.

Binary Analysis

We analyze the Busybox binary across Tomato forks, targeting two vulnerabilities:

- **CVE 2011-2716:** Busybox pre-1.20.0 allows remote servers to execute commands on client side.
- **CVE 2016-2148:** Buffer overflow on client side in Busybox pre-1.25.0 allows remote attackers to have unknown impact.

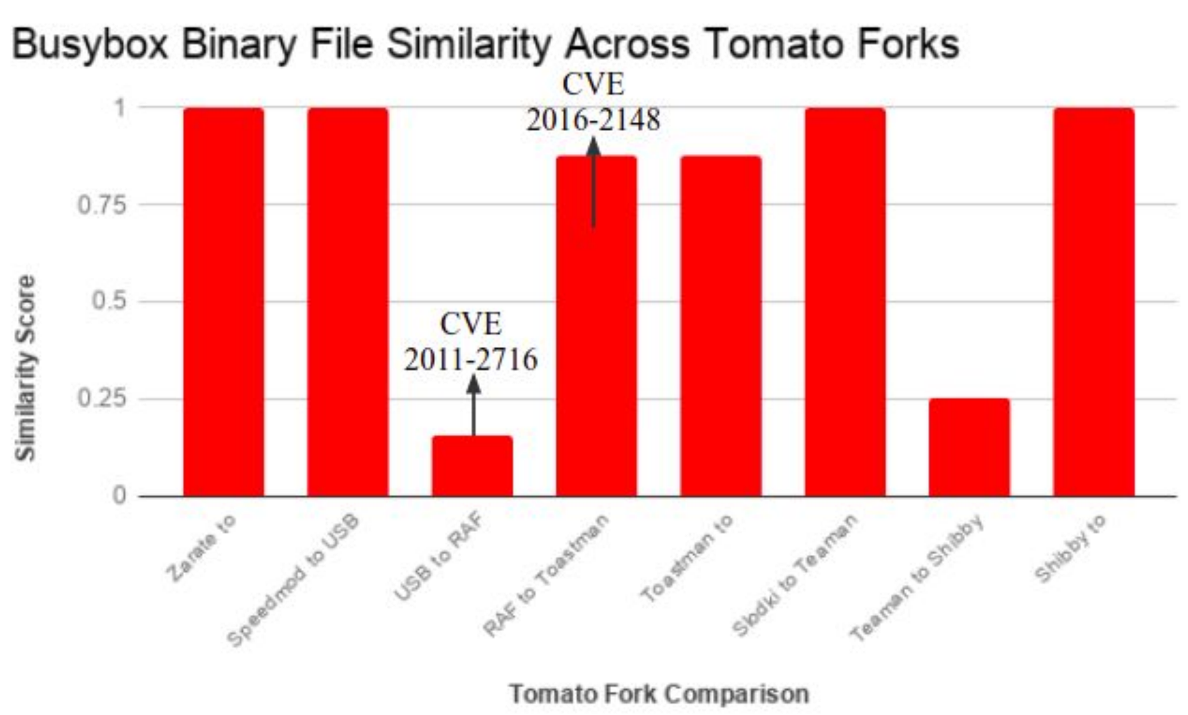
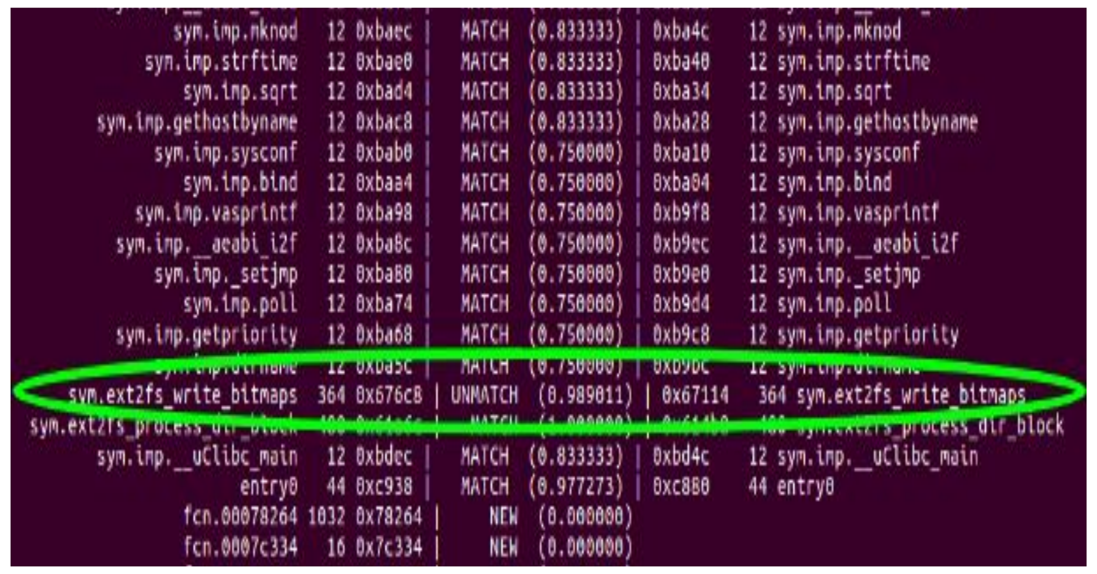


Figure 3: Busybox binary code similarty across Tomato forks.

In Figure 3, we find

- CVE 2011-2716 is patched in USB to RAF with a similarity score of 0.156,
- CVE 2016-2148 is patched in RAF to Toastman with a 0.877 similarity score, and
- Teaman to Shibby scored 0.249, which we disregard due to no no known vulnerability.

We find that USB to RAF is a timely and effective patch of CVE 2011-2716, as functions are new or not a match. We continue to examine CVE 2016-2148 in the RAF to Toastman patch:



```
           sym.imp.mknod   12 0xbaec |   MATCH  (0.833333) | 0xba4c   12 sym.imp.mknod
        sym.imp.strftime   12 0xbae0 |   MATCH  (0.833333) | 0xba40   12 sym.imp.strftime
            sym.imp.sqrt   12 0xbad4 |   MATCH  (0.833333) | 0xba34   12 sym.imp.sqrt
   sym.imp.gethostbyname   12 0xbac8 |   MATCH  (0.833333) | 0xba28   12 sym.imp.gethostbyname
         sym.imp.sysconf   12 0xbab0 |   MATCH  (0.750000) | 0xba10   12 sym.imp.sysconf
            sym.imp.bind   12 0xbaa4 |   MATCH  (0.750000) | 0xba04   12 sym.imp.bind
       sym.imp.vasprintf   12 0xba98 |   MATCH  (0.750000) | 0xb9f8   12 sym.imp.vasprintf
      sym.imp.__aeabi_i2f  12 0xba8c |   MATCH  (0.750000) | 0xb9ec   12 sym.imp.__aeabi_i2f
         sym.imp._setjmp   12 0xba80 |   MATCH  (0.750000) | 0xb9e0   12 sym.imp._setjmp
            sym.imp.poll   12 0xba74 |   MATCH  (0.750000) | 0xb9d4   12 sym.imp.poll
     sym.imp.getpriority   12 0xba68 |   MATCH  (0.750000) | 0xb9c8   12 sym.imp.getpriority
 sym.ext2fs_write_bitmaps  364 0x676c8 | UNMATCH (0.989011) | 0x67114  364 sym.ext2fs_write_bitmaps
     sym.imp.__uClibc_main 12 0xbdec |   MATCH  (0.833333) | 0xbd4c   12 sym.imp.__uClibc_main
                  entry0   44 0xc938 |   MATCH  (0.977273) | 0xc880   44 entry0
           fcn.00078264  1032 0x78264 |     NEW  (0.000000)
           fcn.0007c334    16 0x7c334 |     NEW  (0.000000)
```

Figure 4: Function comparison of Toastman (left) and RAF (right) Busybox binary.

- A highly similar function denoted as ‘not a match’ is what we hope to be a patch.
- In Figure 4, note the sym.ext2fs_write_bitmaps function: it is 98% similar yet it is not a match.
- With Radare2, we are able to get a closer look at the functions using their respective Control Flow Graph (CFG) representations.

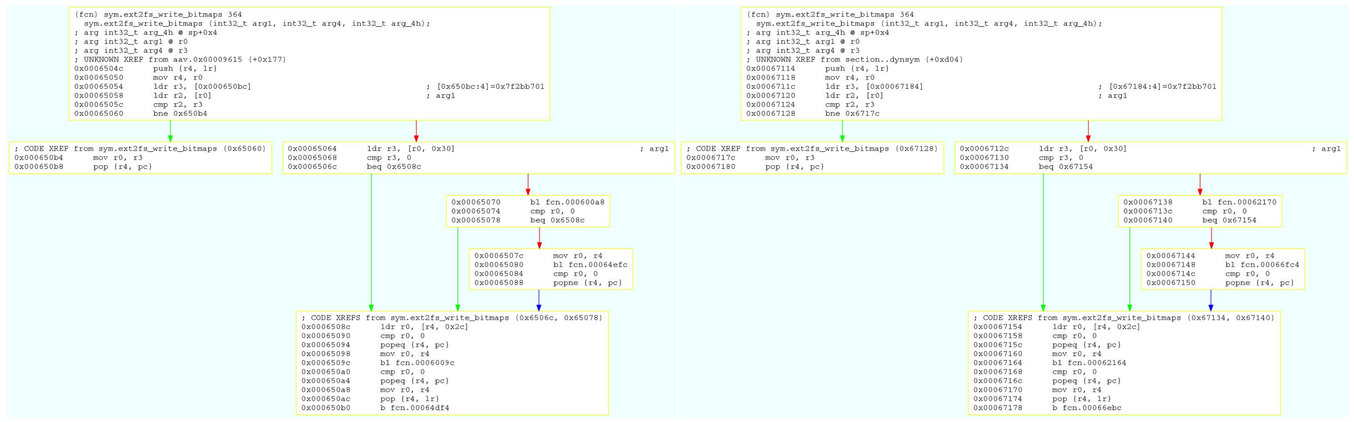
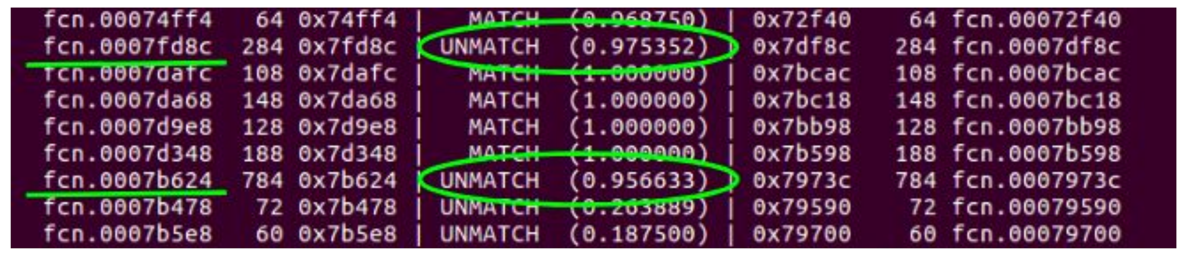


Figure 5: CFG representation of write_bitmaps binary function in Toastman (left) and RAF (right).

- Based on Figure 5 comparison, we must continue to find vulnerable function and patch in the binary.



```
fcn.00074ff4    64 0x74ff4 |   MATCH  (0.968750) | 0x72f40    64 fcn.00072f40
fcn.0007fd8c   284 0x7fd8c | UNMATCH  (0.975352) | 0x7df8c   284 fcn.0007df8c
fcn.0007dafc   108 0x7dafc |   MATCH  (1.000000) | 0x7bcac   108 fcn.0007bcac
fcn.0007da68   148 0x7da68 |   MATCH  (1.000000) | 0x7bc18   148 fcn.0007bc18
fcn.0007d9e8   128 0x7d9e8 |   MATCH  (1.000000) | 0x7bb98   128 fcn.0007bb98
fcn.0007d348   188 0x7d348 |   MATCH  (1.000000) | 0x7b598   188 fcn.0007b598
fcn.0007b624   784 0x7b624 | UNMATCH  (0.956633) | 0x7973c   784 fcn.0007973c
fcn.0007b478    72 0x7b478 | UNMATCH  (0.263889) | 0x79590    72 fcn.00079590
fcn.0007b5e8    60 0x7b5e8 | UNMATCH  (0.187500) | 0x79700    60 fcn.00079700
```

Figure 6: Stripped function comparison of Toastman (left) and RAF (right) Busybox binary.

- Figure 6 shows more functions to examine.
- Though the functions have been stripped, the patch may be found here.

Conclusions

- Continue to find vulnerable code clones and patches.
- Busybox seems to have timely patch in CVE 2011-2716.
- Examine other software projects to find vulnerable code and hold programmers accountable.

References

[1] D. Chen, M. Egele, M. Woo, and D. Brumley, “Towards automated dynamic analysis for linux-based embedded firmware,” in Proceedings of 2016 Network and Distributed System Security Symposium, 2016.

[2] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A large-scale analysis of the security of embedded firmwares,” in Proceedings of 23rd USENIX Security Symposium, 2014, pp. 95–110.

[3] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in Proceedings of 2017 ACM SIGSAC Conference on Computing and Communications Security, 2017, pp. 2201–2215.

Acknowledgements

Support for student stipends, supplies, and/or equipment used in this research was supplied by the Program for Research Initiatives in Science and Math (PRISM) at John Jay College. PRISM is funded by the Title V and the HSI-STEM programs within the U.S. Department of Education; the PAESMEM program through the National Science Foundation; and New York State’s Graduate Research and Technology Initiative and NYS Education Department CSTEP program.