2018 IEEE Symposium on Security and Privacy Workshops

A binary analysis approach to retrofit security in input parsing routines

Jayakrishna Menon[1], Christophe Hauser[1], Yan Shoshitaishvili[2], and Stephen Schwab[1]

[1]*Information Sciences Institute, University of Southern California*
[2]*Arizona State University*

Abstract

In spite of numerous attempts to mitigate memory corruption vulnerabilities in low-level code over the years, those remain the most common vector of software exploitation today. A common cause of such vulnerabilities is the presence of errors in string manipulation, which are often found in input parsers, where the format of input data is verified and eventually converted into an internal program representation. This process, if done manually in an ad-hoc manner, is error prone and easily leads to unsafe and potentially exploitable behavior. While principled approaches to input validation exist, such as those based on parser generators (e.g., Lex [20] and Ragel [28]), these require a formalization of the input grammar, which is not always a straightforward process and tends to dissuade programmers. As a result, a large portion of input parsing routines as found in commodity software is still implemented in an ad-hoc way, causing numerous security issues. We propose to address this problem from a post-development perspective, by targeting software presenting security risks in opaque, closed-source environments where software components have already been deployed and integrated, and where re-implementation is not an option (e.g., as part of an embedded device's proprietary firmware). Our system is able to effectively detect vulnerability patterns in binary software and to retrofit security mechanisms preventing exploitation. In a semi-automated setting, it was able to discover an unknown security bug.

1 Introduction

A large number of applications, and in particular, networking daemons and browsers, are constantly exposed to remote and untrusted input data. A secure parsing and handling of such input data, regardless of the complexity of its semantics, is at the forefront of any network facing application, and in a way, represents the first line of defense. Unfortunately, the process of implementing

DOI 10.1109/SPW.2018.00049

input parsing routines in unsafe languages such as C, which currently remains one of the most used languages for systems and network implementations, is error-prone and leads to a large number of security bugs. Worse yet, even in the case of widely used shared libraries such as *e.g.,* libXML (for which 54 Common Vulnerability Exposure (CVE) entries have been reported and added to the MITRE database between 2003 and 2017[1]), new bugs are found on a regular basis despite the numerous tests and audits performed on such standard software components.

A possible angle to address this problem is to rely on development-stage abstractions providing secure constructs, such as parser generators and parser combinators [7, 20, 28]. While effective, such solutions are only applicable to software for which the source code is available. However, a large number of software products involved in the context of many applications and environments, such as the proprietary firmware of IoT devices, remains closed-source. Yet, such software remains largely exposed to input parsing vulnerabilities. Solutions are needed to retrofit security in these environments.

To this end, we propose to leverage recent advances in the domain of binary program analysis, and in particular, static analysis and symbolic execution, in order to retrofit security in the parsing code of closed-source applications. More specifically, we present an approach capable of statically detecting and mitigating *anti-patterns* corresponding to dangerous practices or common programming errors, which lead to unsafe memory behavior (*e.g.*, the unsafe use of dangerous string operations). While this approach does not guarantee the correctness of the resulting parsers, it detects common pitfalls of parser implementations, and provides mechanisms for patching vulnerable code constructs when no alternative method is applicable. By approaching this problem from a static perspective, we are able to provide mechanisms for retrofitting security at scale with no additional runtime cost. We initially focus on a set of anti-patterns suitable for statically detecting common vulnerabilities present in input parsing code. While each anti-pattern combines program analysis techniques with specialized heuristics, and therefore are intrinsically restricted to a subset of all possible vulnerabilities, our analysis provides sufficient semantic understanding to deploy realistic counter-measures in each situation, which is suitable for enforcement relying on static reassembly mechanisms. As such, this work corresponds to the first steps towards a broader spectrum of analyses tailored for scalable vulnerability discovery and mitigation of parsing errors in binary.

In the remainder of this paper, we present three common classes of programming mistakes leading to memory corruption vulnerabilities, which we abstract in three anti-patterns in Section 2, along with binary program analysis models for detecting those "in the wild", and real-life examples based on existing CVE entries which we used as ground truth for validating our approach. Then, in Section 2.6, we discuss possible techniques for retrofitting security in vulnerable closed-source software. Finally, in Section 3.2, we discuss future work and

[1] `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libxml`

possible alternative approaches.

2 Detecting anti-patterns in binary code

One of the most common errors encountered in C and C++ programs[2] is the lack of sanity checks with respect to buffer sizes when performing operations on arrays in memory, leading to out-of-bound memory access. Because strings are represented in memory as arrays of one byte characters in C, manipulations such as copying strings across memory locations or comparing strings together boil down to memory operations using pointers. Manually dealing with these operations tends to confuse programmers, and it has been the source of numerous memory corruption vulnerabilities for over 30 years.

In this work, we focus on three classes of memory corruption vulnerabilities caused by string manipulation errors, which are commonly found in parsing code. Our approach aims to detect such vulnerabilities in real-world proprietary code for which no source code is available, and in particular, in the firmware of IoT devices as well as embedded devices such as Programmable Logic Controllers (PLC). In order to be realistically deployed on a large scale, our approach relies on a combination of lightweight static analysis and heuristics, along with guided symbolic execution focusing on narrowed-down code paths involving potentially unsafe programmatic constructs. We rely exclusively on architecture agnostic abstractions, as we aim to analyze software and firmware of various embedded devices.

Our initial *anti-pattern* detection articulates around the following three commonly encountered programming errors in string manipulations:

1. Unconstrained input buffer size: the program performs string manipulations on the (user-provided) input buffer without checking its size.

2. Attacker-controlled size: the program may check the size of the input buffer prior to manipulation, but the attacker has control over it (*i.e.,* the program blindly "trusts" user-provided size information).

3. Unchecked termination condition: the program performs string operations on possibly incorrectly terminated stings.

While non-exhaustive, this initial set of *anti-patterns* provides a basis for the detection of typical pitfalls in parsing code. In the remainder of this section, we describe each pattern in more detail.

2.1 Detection approach

Our detection approach is comprised of two phases: it starts with lightweight static analysis in order to first identify the presence of possibly dangerous string

[2]Among other low-level programming languages exposing unsafe constructs to the programmer.

operations, and then symbolically executes the paths of interest involving these dangerous operations in order to confirm the presence of a vulnerability.

We initially compute a Control Flow Graph (CFG) of the program under analysis, and automatically pinpoint known string functions from the standard C library. This initial step assumes that we are dealing with a dynamically linked binary program, which may be stripped (*i.e.,* symbol information such as function names are not available), but for which dynamic linking information is present[3]. In the case of statically linked binaries and binary blobs, function recognition techniques may be used for this purpose, as discussed in Section 3.

From there, we first isolate a subset of the identified string functions which copy data between two (string) buffers. We consider these functions as potential sinks, and track data dependence backwards across function calls for the *source* buffer and (if present) the *size* parameters of these functions. This analysis is based on the construction of a data dependence graph which recovers def-use chains [4] over statements of the disassembled code. We then use this analysis to compute a backward slice through the program, from each of the identified function parameters towards the first encountered user input function (or until no caller function is found, in the case of library functions, in which case we consider the parameters of the topmost function as user input).

Finally, we attempt to detect the size of the output buffer. If the *destination* parameter points to a stack location, we employ a heuristic-based approach to identify its size. We assume that the program has been compiled with stack protection mechanisms which allocate stack buffers before any other local variables[4]. Since multiple buffers may be allocated in the same stack frame, we leverage variable recovery techniques from the angr [1] framework, based on access pattern recognition and forced execution, in order to identify any additional buffer in between our *destination* pointer and the beginning of the stack. Based on this information, we are able to identify the end of the *destination* buffer, and consequently, its size.

At this point, our analysis considers a string operation to be potentially dangerous based on the following conditions:

- C_1: The *source* parameter is dependent on the user input.
- C_2: The pinpointed function is an unsafe variant of string manipulation functions from the standard C library such as `strcpy` or `sprintf`, which keep appending characters to the destination without any size check.
- C_3: The pinpointed function is a safe variant of string manipulation functions from the standard C library, such as `strncpy` or `snprintf` *but* the size parameter depends on user input and is not sanitized.
- C_4: A possibly unsafe sequence of string functions is called (as detailed in §2.4).

[3]This information is required by the loader for dynamic binding, and is therefore present in all dynamically linked binary even after stripping.

[4]This is a common practice in modern compilers which consists in allocating buffers early in order to prevent overriding other variables on the stack in case of overflow.

An operation is considered potentially dangerous if $C_1 \wedge (C_2 \vee C_3 \vee C_4)$ applies. In this case, we proceed with the second phase of analysis.

In the remainder of this section, we describe each variant of dangerous string manipulations modeled in our approach.

2.2 Unconstrained input buffer size

```
1  #define OPT_ERRMAXSTRLEN 1024  /* Fixed buffer len*/
2  #define opt_warn_2(fmt,var1,var2) do { \
3          char gstr[OPT_ERRMAXSTRLEN]; sprintf(gstr,fmt,var1,var2); \
4          opt_warning(gstr); } while(0)
5
6  long opt_atoi(char *s)
7  {
8          int valid;
9          long x;
10         x = (long)atof(s);
11         valid = opt_isvalidnumber(s);
12         if (!valid || (valid & OPT_NUM_FLOAT)) {
13                 opt_warn_2("String [%s] is not a valid integer, will use
                   ↪  [%ld]",s,x);
14         }
15         return x;
16 }
```

Figure 1: *Simplified version of vulnerable code in the* `opt` *input parsing library.*

A straightforward, yet not uncommon case of a "classic" buffer overflow occurs when a string is copied into another, but no verification is made to ensure that the destination string is large enough to contain the source string. Or, put another way, no verification is made to ensure that the input string is not larger than the destination string. The first phase of our analysis detects this case as satisfying $C_1 \wedge C_2$, and yields an input statement s_1 from which user input originates, and a call statement s_2 taking the user input as a parameter to called string manipulation function. It also yields the destination buffer length l_d, as described earlier in Section 2.1.

Upon detection of the above condition, our analysis then executes the resulting program slice forward, from s_1 to s_2. The initial program state is created with an input buffer length l_i that is larger then l_d. If the program correctly checks the length of the input buffer, then the path constraints at s_2 will include the constraint that $l_i \leq l_d$. By definition, since we have initialized l_i to be larger than l_d, this state is unsatisfiable in any program which correctly verifies this condition. Conversely, the presence of a satisfiable state at s_2 indicates improper checking of the input buffer length, and thus the presence of an exploitable buffer overflow.

A real-life example of such a vulnerability is illustrated in Figure 1, which shows a simplified version of vulnerable code present in older versions of the

`opt` open-source input parsing library, upon which many software projects depend in order to parse command line arguments. Unfortunately, all versions of this library prior to, and including, 3.18, were vulnerable to a buffer overflow caused by unsanitized user input. The `opt_atoi` library function takes the user-provided string `s` as input, and the call to the `opt_warn2` macro at line 13 uses that string directly, resulting in an unsafe call to `sprintf` at line 3. If the input string is longer than `OPT_ERRMAXSTRLEN`, a buffer overflow occurs. This bug was reported as CVE-2003-0390[5].

2.3 Attacker controlled size

```
1  int phar_parse_zipfile(php_stream *fp, char *fname, int fname_len, char
   ↪  *alias, int alias_len, phar_archive_data** pphar, char **error)
2  {
3          phar_zip_dir_end locator;
4          char buf[sizeof(locator) + 65536];
5          zend_off_t size;
6          uint16_t i;
7          phar_archive_data *mydata = NULL;
8          phar_entry_info entry = {0};
9
10         mydata = pecalloc(1, sizeof(phar_archive_data), PHAR_G(persist));
11         mydata->fname = pestrndup(fname, fname_len,
           ↪  mydata->is_persistent);
12         mydata->fname_len = fname_len;
13         entry.phar = mydata;
14         phar_set_inode(&entry);
15 }
16
17 static inline void phar_set_inode(phar_entry_info *entry TSRMLS_DC)
18 {
19         char tmp[MAXPATHLEN];
20         int tmp_len;
21
22         tmp_len = entry->filename_len + entry->phar->fname_len;
23         memcpy(tmp, entry->phar->fname, entry->phar->fname_len);
24         memcpy(tmp + entry->phar->fname_len, entry->filename,
           ↪  entry->filename_len);
25         entry->inode = (unsigned short)zend_get_hash_value(tmp, tmp_len);
26 }
```

Figure 2: *Simplified version of vulnerable code in PHP.*

Another common case of a "classic" buffer overflow, which corresponds to a situation which is similar to the one described in § 2.2, is when an attacker directly controls the size of the input buffer. In this case, the program performs a check on the input buffer size, but it trusts a user-provided value to do so, which may lead to a buffer overflow. The first phase of our analysis detects this case as satisfying $C_1 \wedge C_3$, and, similarly to the situation described in §2.2,

[5] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0390`

yields an input statement s_1, and a call statement s_2, as well as the length l_d of the destination buffer. If the condition $C_1 \wedge C_3$ is satisfied, our analysis executes the resulting program slice from s_1 to s_2, and evaluates the constraints on the *size* parameter of the called function. If a constraint exist, such that $size \leq l_d$, then the program performs the appropriate checks. If not, a buffer overflow may be triggered by an attacker. The following is a real-life example of such vulnerability.

Figure 2 shows a simplified version of the code of a vulnerable function in the PHP interpreter. The stack buffer `tmp` of fixed size `MAXPATHLEN` declared at line 19 can be overflown by the subsequent unsafe call to `memcpy` at line 23. The problem here is that the attacker has control over `entry->phar->fname_len`, as this value is directly copied from the user controlled `fname_len` parameter to `phar_parse_zipfile` without sanitization. This bug was reported as CVE-2015-3329[6].

2.4 Unchecked termination condition

```
1  static ssize_t clusterip_proc_write(struct file *file, const char __user
   ↪  *input,
2                  size_t size, loff_t *ofs)
3  {
4  #define PROC_WRITELEN        10
5          char buffer[PROC_WRITELEN+1];
6
7          if (copy_from_user(buffer, input, PROC_WRITELEN))
8                  return -EFAULT;
9
10         if (*buffer == '+')
11                 nodenum = simple_strtoul(buffer+1, NULL, 10);
12 }
```

Figure 3: *Simplified version of vulnerable code in the Linux kernel.*

The lack of correct string termination is also a very common source of errors in C and C++. It is dangerous, because it is commonly assumed that strings terminate with a NULL byte: for instance, standard functions such as `strlen` and `strcpy` rely on this assumption, and lead to unexpected behavior when this condition is not met. Simultaneously, a number of standard functions such as `memcpy` or `strncpy` do not guarantee correct string termination. A direct result of this is that programmers need to be careful when invoking these functions sequentially. The static phase of our approach models such situations as satisfying $C_1 \wedge C_4$. In this case, it would return a statement s_1 representing a call to a function which does not guarantee correct string termination; and a statement s_2 representing a call to a function which assumes the use of a string termination symbol. When the above condition is satisfied, our analysis may

[6] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3329`

have detected unsafe sequence of string operations over a string originating from user input. It then symbolically executes a program slice starting from s_1 and ending in s_2, and evaluates the constraints accumulated on the strings passed as parameters to the second function in s_2. A safe program will ensure that the last byte of such strings is equal to `NULL`. The lack of such constraints when reaching s_2 reveals a vulnerability.

Figure 3 shows the simplified version of a vulnerable kernel function within the NETFILTER subsystem, which does not ensure the presence of a `NULL` character terminating the string obtained from the call to `copy_from_user` at line 7. The consequence of this is the undefined behavior of the call to `simple_strtoul` at line 11, which assumes correct string termination. This bug was reported as CVE-2011-2534[7].

2.5 Evaluation

Our proof of concept implementation leverages the angr [1] framework, on top of which we build our exploration approach and analysis templates. Our initial prototype models two of the three presented anti-patterns: "unconstrained input buffer size", presented in 2.2 and "attacker controlled size", presented in 2.3.

We have evaluated our approach on binary images corresponding to the aforementioned vulnerable versions of libopt and the PHP interpreter. For each binary, a first phase of static analysis pinpoints dangerous functions as a first step, and reasons about data dependence as a second step. These steps are referred to as SA1 and SA2 in the remainder of this section. In cases where the result of (SA1 + SA2) is positive, our approach leverages symbolic execution in order to confirm (or refute) the presence of a vulnerability. In other words, the analyzed code is considered vulnerable if and only if SA1, SA2 and SE all return positive.

Table 1 represents a summary of our analysis results. Each reported function corresponds to a positive match with respect to the first step (SA1) of static analysis. The column GT indicates the ground truth, *i.e.,* whether the code is actually vulnerable. We obtained the ground truth by manually verifying the analysis results[8]. Columns indicating "negative(*)" represent false negatives caused by the data-dependence analysis. In these situations, where SA2 does not align with the observed ground truth, we also executed the vulnerable code paths symbolically.

In addition to this, when analyzing the results of our evaluation where SA2 failed to detect data dependence from user input, we manually pinpointed its location and ran the symbolic step over the resulting code paths. When doing so, our system was able to find a new security bug in the PHP interpreter caused by improper checks on the size of an input buffer (corresponding to the case described in §2.3). We have reported it to the PHP security team and are waiting for an official response before disclosing further details. A

[7] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2534`

[8] These results, however, only report a reduced subset of the functions in the PHP interpreter, due to time constraints, and to the large and complex nature of this library.

Function name	SA1	SA2	SE	GT
(**libopt**)				
opt_atoi	positive	positive	positive	positive
opt_atof	positive	positive	positive	positive
opt_atou	positive	positive	positive	positive
optrega(1)	positive	positive	negative	negative
optrega(2)	positive	negative	-	negative
opt_action	positive	negative	-	negative
opstrval	positive	negative	-	negative
opt_parse_delim	positive	negative	-	negative
opt_from_fname	positive	negative	-	negative
opt_parse_longdelim	positive	negative(*)	positive	positive
optrega_array	positive	negative	-	negative
(**PHP**)				
phar_detect_phar_fname_ext	positive	negative(*)	positive	positive
phar_parse_zipfile	positive	negative(*)	positive	positive
phar_parse_tarfile	positive	negative(*)	positive	positive
phar_parse_tarfile	positive	negative(*)	state explosion	positive
Undisclosed vulnerability	positive	negative(*)	positive	positive

Table 1: *Analysis results*

simplified, anonymized version of the vulnerability is represented in Figure 4. The prototype of `memcpy` expects an unsigned value (of type `size_t`) but in this vulnerable function, the `size` parameter is signed, and therefore the check at line 5 can be bypassed (negative values) which leads to an integer overflow. The patch we submitted adds an extra constraints ensuring that `size > 0`. Despite the presence of this dangerous operation, the actual exploitability of this vulnerability has not yet been confirmed.

Overall, these analysis results demonstrates the practicality of our approach, both in a fully automated and in a semi-automated setup.

```
1  int vulnerable(const char *input, int size)
2  {
3          char output[MAXSIZE];
4
5          if (size >= MAXSIZE-1) {
6                  exit();
7          }
8          memcpy(output, input, size);
9          ...
10 }
```

Figure 4: *Simplified version of a vulnerability in the PHP interpreter*

We observe that SA1 results in false positives in 43.8% of all cases. SA2 is able to filter most false positives (except for 6.6% of those), but introduces false negatives 40% of the time. Finally, symbolic execution is accurate in all cases, except for one function where it reached state explosion. Static analysis execution time (SA1 + SA2) ranged from under 1 second up to 260 seconds. Symbolic execution time ranged from under one second up to 400 seconds (and

over 900 seconds in the case of state explosion).

The accuracy and execution time of our system is summarized in Table 2.

	Static phase (S1 + S2)	**Symbolic phase**	**Overall**
FP	6.6%	0%	0%
FN	40%	*	40%
Time	1-260s	1-400s	2-660s

Table 2: *Overall accuracy and execution time*

The relatively high level of false negatives (*i.e.,* undetected instances of vulnerabilities) in our static analysis approach is currently the bottleneck in terms of accuracy. It is due to several challenges of binary program analysis which hinder data dependence tracking: data structure recovery and pointer aliasing. In all 5 cases where SA2 is inaccurate, complex data structures are used, involving complex access patterns, such as heap structures with multiple levels of dereference, which causes incomplete def-use chains recovery. As a result, our analysis is unable to infer data dependence from user input in these situations. However, it was able to accurately reason about each code path during the symbolic execution phase, when we manually provided the location of the user input, with the exception of one instance of state explosion (which would otherwise be provided by the second step of static analysis). Nonetheless, mechanisms to accurately reason about complex access patterns are required in order to improve the accuracy of the static phase of our approach.

2.6 Mitigation

Our approach, in each detected vulnerable case, employs a constraints-based model in order to reason about the presence of a vulnerability. By relying on symbolic execution of the program under analysis, our model is able to gather accurate constraints over the state of the program. As a result of this accuracy, precise modifications of the program (*i.e.,* patches) can be inferred. It should be emphasized that the lack of accuracy in the static analysis phase does not affect the validity of our overall approach, since, in the presence of a potential vulnerability, symbolic execution is always leveraged in order to precisely reason about the involved program paths (however, our detection phase may miss instances of vulnerabilities, as discussed in §2.5).

Each of the anti-patterns that we described reveal the presence of a vulnerability which can be mitigated by enforcing additional constraints in program paths originating from user input. For instance, in the case described in §2.2, a conceptually simple fix is to add a guard condition regarding the size of the data to copy. This can be achieved, for instance, by replacing the call to `sprintf` with a call to `snprintf`, thus ensuring that the copy will never append data past the boundaries of the destination buffer.

However, despite the conceptual simplicity of such changes, the process of statically modifying programs at the binary level involves a number of chal-

Size (bytes)	0-5	5-10	10-15
Avg occurrences	20.8	24.4	31.5

Table 3: *Function padding in coreutils*

lenges: most of these changes require the insertion of a minimum of three additional instructions preceding each vulnerable *call* instruction. On the one hand, inserting multiple additional instructions within several basic blocks in the code segment of the program will inevitably involve a displacement of instructions, which requires a fine-grained analysis in order to ensure that all subsequent code references are correctly relocated. Failure to do so will break the program. On the other hand, applying such modifications without any side effects beyond a local scope of one to several basic blocks would require the analysis to reorganize and optimize instructions in such a way that additional instructions may be inserted in place. A general approach to solving this problem may leverage recent advances in static reassembly techniques [30, 29]. We plan to investigate this research direction in future work.

In the context of this work, we present a simple alternative which, despite some limitations, is able to effectively mitigate vulnerabilities where minimal modifications are required. We observe that, in most binaries, padding bytes are inserted between functions, for performance reasons (*i.e.,* as produced by GCC's `falign-*` options). Consequently, alignment slots of up to 15 bytes are typically present in binaries. The results of an analysis of all 104 binaries present in the coreutils package on Ubuntu 16.04 is presented in Table 3.

Our approach leverages this unused padding space in order to insert and chain trampoline code gadgets specifically crafted to mitigate known vulnerability patterns. Inserting code in the binary image in this way guarantees that no references to the initial code segment will be broken. While such changes involve a performance penalty caused by cache misalignment, we expect it to be negligible in practice, since most changes are light with few occurrences.

```
1   0x4012b0 push   rbp
2   0x4012b1 push   rbx
3   0x4012b2 xor    esi,esi
4   0x4012b4 mov    rbp,rdi  ;<-- Input
5   0x4012b7 sub    rsp,0x418
6   0x4012eb mov    rdi,rsp
7   0x4012ee mov    r9,rbx
8   0x4012f1 mov    r8,rbp
9   0x4012f4 mov    ecx,0x405cb0
10  0x4012f9 mov    edx,0x400
11  0x4012fe mov    esi,0x1
12  0x401303 xor    eax,eax
13  0x401305 call   0x400f30 <sprintf>
```

Figure 5: *Assembly code of* `opt_atoi` *from libopt*

Figure 5 shows the vulnerable assembly code of the `opt_atoi` function from the libopt option parsing library. Figure 6 shows the patched version of this code. On line 13, a the call to `sprintf` is replaced with a trampoline to code inserted within padding space. The patch introduces four blocks of 11 bytes, 14 bytes, 5 bytes and 8 bytes respectively. The first inserted block starting at 0x402991 saves registers on the stack. The second block starting 0x4029d1 checks the size of the input string and either jumps to the block at 0x402a44 to terminate the program if it is larger than the destination buffer, or jumps to the block at 0x402a11 to restores the registers and calls `sprintf`.

```
1      0x4012b0 push   rbp
2      0x4012b1 push   rbx
3      0x4012b2 xor    esi,esi
4      0x4012b4 mov    rbp,rdi  ;<-- Input
5      0x4012b7 sub    rsp,0x418
6      0x4012eb mov    rdi,rsp
7      0x4012ee mov    r9,rbx
8      0x4012f1 mov    r8,rbp ; <-- Input
9      0x4012f4 mov    ecx,0x405cb0
10     0x4012f9 mov    edx,0x400
11     0x4012fe mov    esi,0x1
12     0x401303 xor    eax,eax
13     0x401305 call   0x402991  <-- Trampoline
14   -------
15     0x402991 push   rdi
16     0x402992 push   rsi
17     0x402993 push   rdx
18     0x402994 push   rcx
19     0x402995 push   r8
20     0x402997 pop    rdi
21     0x402998 push   r8
22     0x40299a push   r9
23     0x40299c jmp    0x4029d1
24   -------
25     0x4029d1 call   0x400d90 <strlen>
26     0x4029d6 cmp    rax,0x400
27     0x4029dc jbe    0x402a11
28     0x4029de jmp    0x402a44
29   -------
30     0x402a44 call   0x400ef0 <exit>
31   -------
32     0x402a11 pop    r9
33     0x402a13 pop    r8
34     0x402a15 pop    rcx
35     0x402a16 pop    rdx
36     0x402a17 pop    rsi
37     0x402a18 pop    rdi
38     0x402a19 call   0x400f30 <sprintf>
```

Figure 6: *Patched assembly code of* `opt_atoi` *from libopt*

In essence, this mitigation is equivalent to the verification and protection routines inserted by recent compilers, such as `__sprintf_chk`, which terminate the program when a buffer overflow is detected. This technique makes it possible to retrofit a similar mechanisms within legacy binaries or code compiled without such protections.

3 Discussion

We demonstrate that leveraging binary program analysis techniques combined with vulnerability models can be effective in a static setting, *i.e.,* our system does not require concrete execution (*i.e.,* dynamic analysis), and only relies on selective symbolic execution of restricted code paths flagged by lightweight static analysis. An advantage of this approach is its lightweight aspect, and therefore its ability to scale. In future work, we plan to leverage it in the context of large corpora of binary program images, and, in particular, firmware images obtained from IoT devices, for which automated reasoning is currently limited to high-level vulnerabilities [10] such as those present in remote web configuration interfaces [6, 11] or specific classes of logic bugs [24].

The approach described in this work addresses the problem of detecting and mitigating input parsing errors in *unknown* binary code. It has the advantage of being agnostic to the actual parser's semantics (with respect to *i.e.,* the protocol or file format that it operates on), but it does not guarantee soundness. However, our system generates and applies patches in a conservative way. The patches generated from symbolic input constraints are accurate. Additionally, if our system is unable to reason about a program path, it will stop and proceed with the remaining program paths to analyze. While this approach does not provide the same level of semantic reasoning as source-code level approaches, it is effective as a last resort defense in environments where no or little information is available to the analyst.

3.1 Limitations

Our approach does not currently support the recovery of complex structures, nor object field sensitivity. A consequence of this is that the resulting data-flow analysis is not accurate, which leads to the presence of an important rate of false negatives (40%) in our static analysis. During the second phase, *i.e.,* the process of symbolically executing the identified program slices yields accurate results and expressions, and effectively filters false positives down to 0%. As a consequence, our overall approach is subject to false negatives during vulnerability discovery, on top of which we apply a conservative mitigation strategy. Our initial focus is on the detection of stack-based buffer overflows, where the destination buffer is allocated on the stack. The buffer recognition technique defined earlier in §2.1 is limited to analyzing memory access patterns within a given stack frame, based on a static memory model. We do not currently address the problem of detecting the size of heap-allocated buffers, which requires a more accurate

static memory model, and which we leave for future work.

3.2 Future work

In future work, we project to extend the scope of our system to a larger set of vulnerability models, for deployment on large scale analysis of embedded firmware images. For a more fine-grained static reasoning, an improved data-flow model is required in order to track def-use chains on long program paths. Of particular interest in this space are techniques for heuristic-based data structure recovery, in combination with more accurate memory models, such as the abstract model provided by Value Set Analysis, among other possibilities.

In a number of situations, some components may actually be recognized, or may be known in advance by the analyst. In such cases, a possible approach to guarantee the correctness of a parser's implementation within a proprietary software component is to replace it with a known-correct one. This may not always be possible, since the semantics of the parser itself might not be known in advance. However, it is common for proprietary code to embed open-source libraries or components. Unfortunately, such components or libraries are not always updated within the remaining lifetime of the project, thus exposing the final software product to vulnerabilities which are likely to be readily discovered and exploited through the analysis of software repository change logs. When such a situation occurs, solutions based on static reassembly may be leveraged for replacing the library with a newer, safer version, or to replace a component's parser with a proven correct one.

4 Related Work

Efforts to detect and mitigate memory corruption bugs in low-level programming languages are not new [27, 13, 3, 2]. For over 35 years, attackers have been able to hijack the control flow of programs written in languages such as C and C++, which remain two of the most used programming languages today, especially in embedded environments where performance is critical. Past and recent research efforts to address this problem include fuzzing-based techniques [26, 22, 16, 9, 17], program transformation and compiler-based techniques [15, 8, 14] and static techniques [31, 19, 12, 18, 5], among others.

This preliminary work builds on existing techniques from the domain of binary program analysis [25, 24], combined with heuristics tailored for the detection of multiple facets of input parsing errors leading to unsafe string manipulations in C and C++ programs. Our approach does not require the source code of the analyzed application, does not require full system emulation or specific hardware, and has the potential to apply at scale to the software and firmware of multiple architectures.

Piston [23], takes a different approach to (uncooperative) patching relying on binary diffing in order to identify and replace known functions in the target binary. Our work is orthogonal to this approach, and applies to unknown

functions as well.

In recent years, mitigation techniques against memory corruption vulnerabilities were added to most of the major operating systems and compilers. Those include stack canaries [13], address space layout randomization (ASLR) [3], write or execute (also known as $W \oplus X$) [2], XOM [21] among others. While these techniques are efficient at mitigating attacks by raising the bar for attackers to successfully exploit vulnerabilities, they do not fix the root cause of such vulnerabilities: the fact that a vast amount of programs are still written in unsafe languages such as C and C++, and that memory corruption bugs remain prominent in commodity software.

References

[1] angr, a binary analysis framework. `http://angr.io`.

[2] OpenBSD's W^X. `http://www.openbsd.org/papers/bsdcan04/mgp00005.txt`.

[3] The PAX Team. `https://pax.grsecurity.net`.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[5] G. Chatzieleftheriou and P. Katsaros. Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 96–103, July 2011.

[6] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[7] Pierre Chifflier and Geoffroy Couprie. Writing parsers like it is 2017. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 80–92. IEEE, 2017.

[8] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 409–417. IEEE, 2001.

[9] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, pages 95–110, 2014.

[11] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448. ACM, 2016.

[12] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 269–278. IEEE, 2006.

[13] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[15] Christopher Dahn and Spiros Mancoridis. Using program transformation to secure c programs against buffer overflows. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 323–332. IEEE, 2003.

[16] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[17] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[18] Roland Kindermann. Static Detection of Buffer Overflows in Executables. 2008.

[19] David Larochelle, David Evans, and others. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, volume 32. Washington DC, 2001.

[20] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc.* " O'Reilly Media, Inc.", 1992.

[21] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[22] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Piston: Uncooperative remote runtime patching. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 141–153. ACM, 2017.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[25] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[26] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[27] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[28] Adrian Thurston. *Ragel state machine compiler*. 2003.

[29] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[30] Shuai Wang, Pei Wang, and Dinghao Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 236–247. IEEE, 2016.

[31] Misha Zitser, Richard Lippmann, and Tim Leek. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 97–106, New York, NY, USA, 2004. ACM.