Armor Within: Defending against Vulnerabilities in Third-Party Libraries

Sameed Ali
Department of Computer Science
Dartmouth College
Hanover, USA
sameed.ali.gr@dartmouth.edu

Prashant Anantharaman
Department of Computer Science
Dartmouth College
Hanover, USA
pa@cs.dartmouth.edu

Sean W. Smith
Department of Computer Science
Dartmouth College
Hanover, USA
sws@cs.dartmouth.edu

***Abstract*—Vulnerabilities in third-party software modules have resulted in severe security flaws, including remote code execution and denial of service. However, current approaches to securing such libraries suffer from one of two problems. First, they do not perform sufficiently well to be applicable in practice and incur high CPU and memory overheads. Second, they are also harder to apply to legacy and proprietary systems when the source code of the application is not available. There is, therefore, a dire need to secure the internal boundaries *within* an application to ensure vulnerable software modules are not exploitable via crafted input attacks.**

We present a novel approach to secure third-party software modules without requiring access to the source code of the program. First, using the foundations of *language-theoretic security*, we build a validation filter for the vulnerable module. Using the foundations of linking and loading, we present two different ways to insert that filter between the main code and the vulnerable module. Finally, using the foundations of *ELF-based access control*, we ensure any entry into the vulnerable module must first go through the filter.

We evaluate our approaches using three known real-world exploits in two popular libraries—libpng and libxml. We were able to successfully prevent all three exploits from executing.

***Keywords*-Language-theoretic security, buffer overflows, input-handling vulnerabilities, access control**

I. Introduction

As software grows complex, the use of libraries and other such third-party software modules has become ubiquitous. The extensive use of such software modules presents a difficult challenge for software security as vulnerabilities found in third-party software modules can compromise the security of the entire program, as it's all within the same process address space.

Software exploits occur when an unexpected input causes the program to undergo an unexpected program execution path, which leads to undesired program behavior. In prior work, this behavior has also been described as the discovery and programming of ”weird machines” [1]. However, looking at recently discovered vulnerabilities, we notice that a significant number are due to crafted input exploits against libraries.[1]

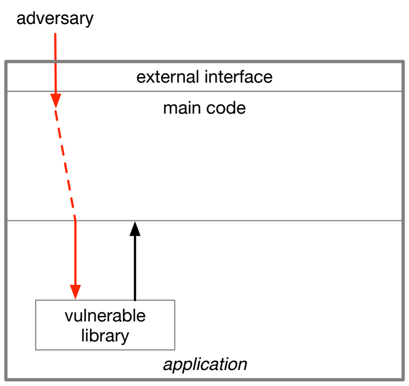

Fig. 1. In the general model we consider, an internal library has a crafted input vulnerability; we must defend against the adversary tricking the main program into calling that library with suitably crafted input.

The standard way to defend against crafted input attacks is validate the input that comes in at the adversary's attack surface. However, when an internal library is vulnerable (Figure 1), it's not clear what to filter for at the adversary's attack surface. We therefore propose enforcing strong protections at the software module boundaries—e.g., between the main program and each library it uses. These protections will provide a strong guarantee that even if a third-party library is vulnerable to a crafted input attack, the overall application will not be exploited; the adversary may try to trick the main program into passing the right crafted input to the vulnerable library, but our defense will ensure that no unvalidated input crosses that channel. This allows us to reason strongly about software security. We can be sure that any crafted input attack which requires the input to be syntactically invalid will not be able to exploit the software module—thus reducing the severity of another libpng-like 0-day [2], if it were to happen again. Furthermore, these drop-in filters could easily be added to applications after a vulnerability is discovered to protect them from being exploited by a malicious attacker.

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0179. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

[1]We searched http://cve.mitre.org for recent crafted input attacks using the keywords “buffer overflows,” “crafted,” and “heap overflow.”

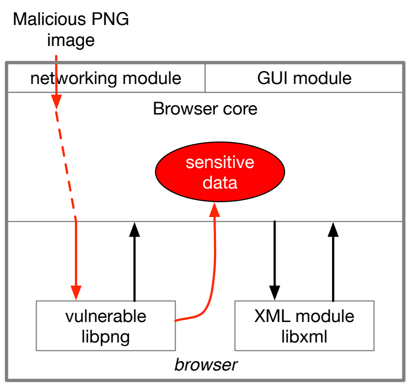

Fig. 2. In CVE-2004-0597, the adversary tricks the browser into sending a malicious PNG file into the libpng library. The exploited software module can then access sensitive information in other parts of the address space.

a) Attacks: Third-party libraries are used by multiple widely used software programs and are often targeted by malicious actors who try to find exploits in these widely used modules so they can reach a larger target audience for their malware.

Programs accept arbitrary input from users, sockets or files. These programs are expected to fail safely and reject malicious inputs. However, in the case of the remote code execution bug found in libpng (CVE-2004-0597), if the adversary can trick the main program into passing a specially crafted PNG to the library, the library overwrites the stack of the main program. Figure 2 visualizes the data-flow of this attack. Traditional security considers hardening the outer boundary of the main program, and treats libraries as black boxes; preventing these attacks requires hardening the boundaries between the main program and the libraries.

b) Existing Defenses: Most solutions that have been proposed—such as Write-XOR-Execute [3], stack cookies [4], DEP [5], SafeSEH [6], CFI [7], and ASLR [8]—do not prevent malformed input from being consumed. Indeed, they attempt to prevent exploits *after* the exploit starts execution by detecting a memory corruption or a control flow integrity violation. Although these techniques do manage to stop some exploits midway in execution, they offer no protections that crafted or malformed input will not be consumed by the application in the first place. In contrast, our drop-in LangSec filters will detect and prevent a crafted input attack before the exploit executes itself. Moreover, sophisticated exploits are often able to bypass these defense techniques, so there is a need for a more effective exploit mitigation mechanism.

The existing software hardening approaches also do not take advantage of software modularity. A defensive mechanism which attempts to use this aspect of software architecture to its advantage will prove highly useful as we shall demonstrate with our approach.

c) This Paper: We tackle these issues using a suite of techniques to provide *Armor Within* an application.

First, we propose enforcing strong protections at software module boundaries. We do this by injecting LangSec filter before data enters the untrusted modules. A LangSec filter is the software component which validates the input by parsing it as a formal language.

Second, we use ELFbac to compartmentalize software and enforce fine-grained access control on code and data sections and across software module boundaries. By ensuring that programs do not jump to code sections that are not allowed via policy, ELFbac enforces that entries to a vulnerable library must first go through the filter.

Finally, we convert legacy binaries to include ELFbac meta-data and LangSec filters to reduce the human effort required to deploy our tools. Our LangSec filters consume CPU time in the order of micro-seconds.

To this end, we make the following contributions:

- We provide a novel technique to prevent exploitation of untrusted libraries by using lightweight LangSec parsers and filters.
- We demonstrate two novel techniques to inject these parsers into an application using these libraries
- We demonstrate a novel use of ELFbac to ensure an attacker cannot bypass these parses
- We base these defense on the scientific foundations of language theory, access control, and linking and loading.

The remainder of the paper is organized as follows: Section 2 provides a recap of LangSec and ELFbac, Section 3 describes our solutions to tackling the problem of insecure libraries, Section 4 discusses strategies to evaluate *Armor Within*, related work is in Section 5, and Section 6 concludes.

II. Background

A. A Recap of LangSec

Any application which takes an input and uses it must somehow *parse* that input. Consuming input that has not been correctly parsed and validated is a primary cause of software vulnerabilities.

The Language-theoretic security (LangSec) approach is to model the input of the program as a formal language, and then validate that inputs lie within this language. This approach applies the *scientific foundations* of formal language theory to input validation. To enable building these validators, the LangSec community has built hardened parser construction libraries—the defending programmer then transforms the language grammar into calls to these combinators. By modeling the input as a formal grammar and validating it, LangSec ensures that the application only consumes syntactically valid input. We chose the Hammer [9] combinator library because of ease of use, but our techniques can be applied using other parser-combinators such as EverParse [10] or Nom [11].

The Hammer parser-combinator library supports various parsing backends to parse regular grammars, context-free grammars, as well as parsing expression grammars. Hammer[2] makes use of combinators and primitives such as these:

- `h_uint8()` — to parse an 8 bit integer.

[2]Hammer supports of a lot of other combinators and primitives, a complete list can be found here [9].

- `h_ch('a')` — to parse the character 'a'.
- `h_length_value(l, p)` — apply parser `l` which returns a number n, then apply parser `p`, n times.
- `h_int_range` — check if the integer parsed lies in the range specified.
- `h_attr_bool` — programmer can provide a custom function that validates the input.

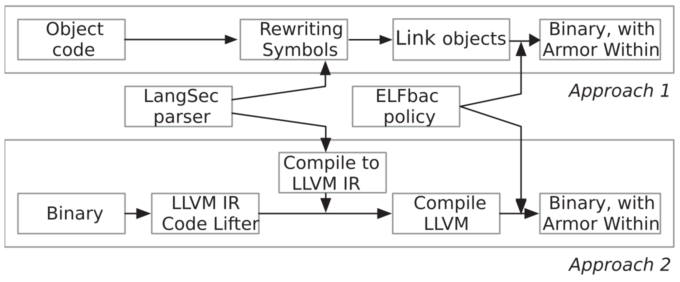

Fig. 3. Overall Architecture of *Armor Within*. The diagram shows the two approaches used to insert LangSec filters to harden a binary.

In this past, many researchers (including our team) have used LangSec to harden applications by placing LangSec filters *on the outer perimeter.* In this paper, we use LangSec *within an application*—at the software module boundaries to secure vulnerable software modules.

B. A Recap of ELFbac

The standard OS security model treats a process address space as the basic unit. All code within the address space is the same subject; writable data in the space can be written by any code in the space.

Our lab's ELFbac approach [12] uses the executable and linking format (ELF) to decompose the code and data inside an address space into smaller units, and then uses a custom kernel patch to provide fine-grained memory access protections among these units.

Prior work has used ELFbac as a mitigation technique to prevent exploits where exploited code tries to inappropriately access data. In this paper, we also use ELFbac to provide limited control-flow integrity, to insure that any calls or jumps into a specific code block must first pass through an interposer.

This policy is added to the ELF binary file. When the OS loads the binary, the loader reads the policy contained in the binary file and enforces it [13].

We use a custom tool to write the policy to the ELF binary file, called `mithril`. To add a policy, first the policy is defined in a ruby file as shown in Listing 3. The file is then read by `mithril` and injected into the ELF binary in a separate ".elfbac" section.

The policy must specify a list of states, the regions of memory that should be executable, along with the regions of data that should be accessible from each particular state. The policy must also specify a starting state, which is the first state the program will be in when control is transferred to it by the loader. The policy allows definitions of tags which allow to refer to separate parts of memory as one group for ease of writing the policy. As an example, the tag for `libc` library is defined in Listing 3 on line 3. The policy has constructs to easily reference parts of the loaded application as well. For instance, to reference parts of ELF binary sections, one can use `section_start` construct and to refer to the code region of a dynamically loaded library one can use the `library_code` construct. The policy also specifies fine-grained permissions on sections such as `readwrite` and `executable`.

III. Proposed Solution

As shown in Figure 4, we propose to secure vulnerable software modules from crafted input attacks by inserting LangSec input validation filters between the main program and the vulnerable module. This binary hardening approach addresses and prevents input validation vulnerabilities present in libraries.

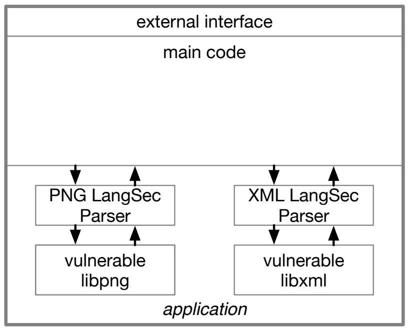

Fig. 4. Placement of LangSec Validation Filters. In a binary that uses both libPNG and libXML, two separate parsers would have to be placed across the boundaries between the binary and the two different libraries.

LangSec filters have been used to effectively prevent crafted input exploits [14] in the past. Inspired by the libpng vulnerability, we extend the prior work in the LangSec space to build these LangSec-compliant parsers for popular file formats, demonstrate that LangSec validation filters are an effective way to secure vulnerable third-party software modules that consume these formats, and present two novel techniques of injecting these parsers in existing legacy binaries.

Furthermore, our approach demonstrates that it is possible to insert the filters mentioned above without access to the source code of the program. However, to build prototypes of *Armor Within*, we assume that the parser-combinator toolkit we use, Hammer, is bug free. We also assume that the control-flow integrity techniques we use in our paper function correctly. An outline of the approaches we used is shown in Figure 3.

A. Basic Model

An attacker trying to compromise a system may have offline-read access to the address space of the binary, but still may not be able to find vulnerabilities to gain access to a shell. The address space may, however, use libraries such as libpng, that could have vulnerabilities.

We assume that the attacker will try to manipulate the main program in order to force a call or jump to an insecure module and feed it the appropriate crafted input. To defend against these attacks, we

- insert LangSec validation filters around the vulnerable internal module,
- enforce that adversary-compromised code goes through these filters before reaching the module

Below, we detail two methods to do this.

B. Method one: Object rewriting

Our first method directly edits the ELF binary.

ELF binaries contain code and data. They also contain a symbol table to keep track of what functions are in an ELF file and at what offsets in the binary they are present. For the first method, our intuition is that we can rewrite this symbol table to rewrite a function offset, and force the binary to run its input through a parser.

This approach assumes we have separate object files for each of the software modules. By manipulating the object files instead of the source code, we are able to inject our parsers and show that the technique works well.

The symbol table, as its name suggests, stores the symbolic references of different modules in the ELF file. It allows the linker to handle symbolic references from one module to another. Before linking the object files together, the symbols are representing the functions in the vulnerable library object file are rewritten to point to the LangSec filter functions. Thus when the linker links the object files together, it will insert the LangSec filter in-place of the vulnerable functions. This filter can then call the vulnerable function after validating the data.

```
Symbol table '.symtab' contains 31 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS readpng.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT   12
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT   14
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT   16
    13: 0000000000000000     0 SECTION LOCAL  DEFAULT   18
    14: 0000000000000000     0 SECTION LOCAL  DEFAULT   19
    15: 0000000000000000     0 SECTION LOCAL  DEFAULT   17
    16: 0000000000000000   337 FUNC    GLOBAL DEFAULT    1 readpng
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND fopen
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_create_read_struct
    20: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND fclose
    21: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_create_info_struct
    22: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_destroy_read_struct
    23: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _setjmp
    24: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_init_io
    25: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_set_sig_bytes
    26: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND wpng_read_png
    27: 0000000000001000   122 FUNC    GLOBAL DEFAULT    1 main
    28: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND stderr
    29: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __fprintf_chk
    30: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND fwrite
```

Listing 1. The symbol table of an object file printed by `readelf`

If a software module were to call a function from another module, the linker will lookup the symbol table, find the corresponding entry for that called function and thus know which part of the code section to jump to execute that function or which dynamic library to load to continue program execution. If the symbol table entry, corresponding to the vulnerable library function, is rewritten to point to the LangSec validation filter instead, the LangSec validation filter function will be executed every time a call to the vulnerable library function is made. Listing 1 shows an example of a symbol table.

The LangSec validation filter validates the arguments passed to the library filter and invokes the library function if the LangSec validation is successful. If the input is malformed, it halts the program. A programmer has the option to specify what to do instead of stopping.

By only executing the vulnerable function of a library after validating the input data via LangSec, we ensure the library function gets syntactically well-formed input. An active adversary cannot exploit vulnerabilities in the library to gain control of the address space, since the LangSec validation filter rejects the malformed input.

To ensure that the adversary cannot get the main program to bypass the filters, we assume the application includes ELFbac-compliant metadata (viz. code and data sections are separate) and we run it on an ELFbac-enhanced kernel (fine-grained permissions are applied and enforced). As Figure 5 shows, we use ELFbac to ensure that the attacker would not be able to jump from anywhere into the vulnerable module, but from particular entry points.

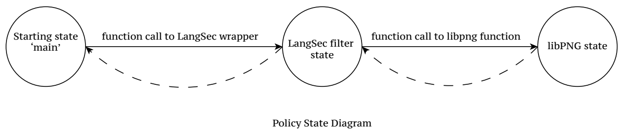

Fig. 5. In ELFbac, the kernel enforces memory access controls based on an FSM and policy we specify. This example shows how libpng is not executable unless the system has entered our LangSec filter. (For simplicity, we do not show the case of when libpng may itself then call other libraries.). The dotted lines represent the state transitions occuring when a function returns.

After rewriting the symbol table of the binary, ELFbac policy is added to it to ensure that there are no direct jumps allowed from the user code directly to the vulnerable library, i.e., a jump bypassing the inserted LangSec filter, as shown in Figure 6. The policy also ensures that the library function is allowed only the data that is relevant to its use.

C. Method two: Binary rewriting

The method of Section III-B has disadvantages.

- This first method needs access to object files (not just the raw binary) to rewrite the symbols. In many cases, This may not be realistic

- It is not effective when binaries are statically linked. Statically linked binaries include entire libraries in their binaries, and hence a lot more addresses and symbols would have to be rewritten, making this approach infeasible to scale.
- Rewriting symbols is also not useful when legacy binaries require additional metadata. If functions and data are not already in separate, self-contained sections, it can be non-trivial to add these metadata to a legacy binary without breaking functionality.

To overcome these limitations, we developed a second technique where we lift the binary to LLVM bitcode and insert the filter as an LLVM IR pass.

We convert legacy binaries to LLVM IR using off the shelf tools such as McSema [15]. McSema lifts binaries to LLVM bitcode via a two-step process. It first gathers the control-flow graph of the program using tools such as IDA Pro, and then it converts this control-flow graph into LLVM bitcode. They were able to demonstrate effectiveness across several architectures.

After obtaining the LLVM bitcode, a custom LLVM IR pass inserts the LangSec filter in the required positions. First, this function scans the LLVM bitcode to locate all the entry points from the user code to a library module. We then examine the list of entry points and map them to the library module to evaluate which parser to inject.

Second, the LLVM IR pass replaces the library function calls in the user code with function calls to the LangSec filters. An example of this can be found in Listing 2. These filters exhaustively validate the input received to make sure they comply with the formal specification of the data format. In our examples, we used the PNG image format and a custom XML format. After validating the data syntax, the filter would then go on to call the library module.

To maintain control-flow integrity, and to prevent attackers from jumping past the LangSec validation filters, we want to insert metadata that ELFbac can use to ensure that the program corresponds to the state-machine policy. *Armor Within* adds ELFbac metadata in a two-step process. First, it adds section information to all the functions and global data. Second, when the LLVM bitcode is compiled back to a binary, it adds a ".elfbac" section to the binary that contains details of the policies that need to be enforced.

D. Implementation

We implemented parsers for PNG and a custom XML format using the Hammer parser-combinator toolkit. Our parsers are written in C. The parser for our PNG header format is in Listing 2. We also implemented an ELFbac policy for the binary that uses the `libpng` library. The policy uses a Ruby-based domain-specific language, and is in Listing 3.

We also implemented method one: symbol rewriting, and method two: binary rewriting. Our symbol rewriting implementation makes use of `objcopy` to edit object files and add sections to it. It then recompiles the object files again to create the final binary with the injected parser.

```c
#include <hammer.h>

HParser* PNG_IHDR_parser() {
    HParser* MAGIC = h_token("\x89\x50\x4E\x47\x0D\x0A\x1A\x0A", 8);
    HParser* IHDR_len = h_int_range(h_uint32(), 13, 13);
    HParser* IHDR_type = h_token("IHDR", 4);

    HParser* width = h_uint32();
    HParser* height = h_uint32();
    HParser* attributes = h_repeat_n(h_int8(), 5);
    HParser* IHDR_chunk = h_sequence(width, height, attributes, NULL);

    HParser* IHDR_crc = h_attr_bool(h_uint32(), validate_crc, NULL);

    HParser* IHDR = h_sequence(MAGIC, IHDR_len, IHDR_type, IHDR_chunk, IHDR_crc, NULL);
    return IHDR;
}
```

Listing 2. Hammer implementation of the PNG header. The PNG header format starts with 8 magic bytes, followed by a 13 byte length field. We use the `h_attr_bool()` function to validate the checksum.

Our second implementation, makes use of an Clang-LLVM IR pass. The binary is lifted using McSema [15]. The lifted binary is run through the Clang-LLVM IR pass. The IR pass is written in C++, and must be compiled using clang to work.

E. Limitations

We are aware that our proposed defenses have certain limitations. First, LangSec parsers offer no protection against crafted input attacks whose payload is *syntactically valid*, but semantically invalid. Dealing with such attacks, however, is outside the scope of this paper.

Second, our defenses require the parsing to be done twice—once in our filter, and then again inside the vulnerable module. In our current approach, the abstract syntax tree generated by the Hammer-based LangSec validator is not utilized by the program's application logic. However, this limitation can be overcome if the application is programmed to use the AST generated by the LangSec filter instead.

Third, our approach requires the programmer to write the LangSec parser for each input format going into the third party library—or at least of the input formats discovered to trigger vulnerabilities. Although it is possible to automate the generation of such parsers with a domain-specific language or a data-description language, it is necessary extra effort required from the programmer.

F. Summary

In summary, our approaches to inject LangSec validation filters comprise rewriting symbols in compiled binaries and lifting binaries to LLVM bitcode and injecting code in the LLVM bitcode. We augment our code injection techniques by using control-flow integrity approaches such as ELFbac. ELFbac places the library modules in their separate states and provides a clear interface to inject our LangSec validation

```
# Tags definition
# for brevity all used tags are not shown
tag :libc do
  library_code('libc.so.6')
  library_code('ld-linux-x86-64.so.2')
end

# the start state
state 'main' do
  exec :libc # libc tag region is executable
  exec :plt
  exec :wpng_init_io
  exec :main
  # readwrite permission for "default" tag region
  readwrite :default
  to 'langsec_filter' do
    # when 'wpng_init_io' function called
    # do transition to LangSec state
    call 'wpng_init_io'
  end
  # transition to libc
  to 'libc' do
    call '_dl_runtime_resolve'
  end
end

# state representing libc
state 'libc' do
  exec :libc
  exec :plt
  readwrite :default
  to 'main' do
    call section_start('.fini')
  end
end

# the LangSec filter state
state 'langsec_filter' do
  exec :wpng_init_io
  exec :plt
  exec :default
  readwrite :default
  to 'libc' do
   # required transition for libc functions
    call '_dl_runtime_resolve'
    call 'malloc'
  end
  # transition to libPNG state
  to 'png_init_io' do
    call 'png_init_io'
  end
end

# the libPNG state
state 'png_init_io' do
  exec :png_init_io
  exec :libc
  exec :plt
  readwrite :default
  # libPNG needs access libc
  to 'libc' do
    call '_dl_runtime_resolve'
  end
end
```

Listing 3. ELFbac policy for libpng. The policy shows four states: main, LangSec filter, libpng, and libc. Each state also clearly specifies what data and code sections it can access, and what states it can transition to, when calling which functions.

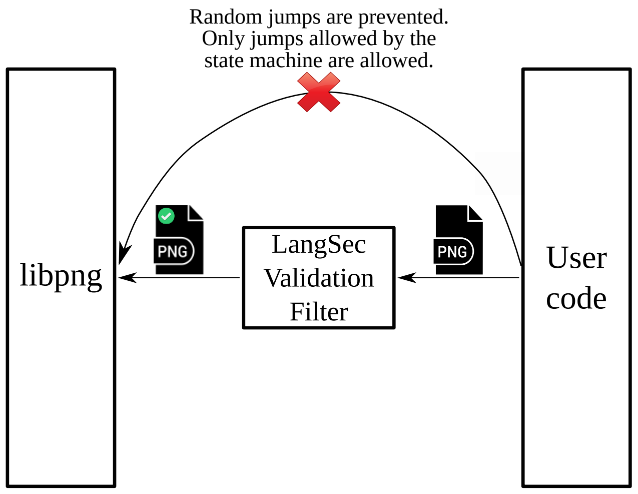
Random jumps are prevented.
Only jumps allowed by the
state machine are allowed.
libpng
PNG
LangSec Validation Filter
PNG
User code

Fig. 6. How ELFbac and LangSec interact with each other. ELFbac prevents random jumps and only allows jumps from specific locations that are clearly specified in the state machine of the binary.

filters. We evaluate the performance of our approaches and discuss challenges in the oncoming sections.

IV. Evaluation and Results

To evaluate our system, we answer the following questions:

- Is *Armor Within* effective against known vulnerabilities?
- How much overhead do our LangSec filters add to existing binaries?
- Can *Armor Within* effectively inject parsers in existing binaries?

A. *Effectiveness against known vulnerabilities*

We reconstructed three known exploits of which two were in the `libpng` library and one was in `libxml` library. We tested the effectiveness of our method by inserting drop-in LangSec validation filters in the vulnerable application ELF binaries. We tried the experiments with the object rewriting approach.

The CVE numbers of the tested exploits as as follows:

- CVE-2016-1838: Denial-of-service heap-based buffer over-read vulnerability in LIBXML
- CVE-2004-0597: Stack-Overflow remote code execution vulnerability in LibPNG
- CVE-2010-1205: Buffer overflow in LibPNG

We ran these experiments on a Desktop computer equipped with a Xeon E3-1245 processor and 8 Gigabytes of RAM. The computer ran Ubuntu Linux version 12.04 with the ELFbac Linux kernel patch.

In all cases, the filters detected the malicious input and prevented the application from being exploited. Both the exploits can be mitigated by using the same LangSec implementation of the PNG parser. All the three vulnerabilities were patched using our filters. This result shows that our method of hardening third-party modules in binaries using a hardened parser-combinator library can effectively prevent crafted input attacks.

PNG file size	No instrumentation	With LangSec filter	LangSec filter with ELFbac policy
100KB			
Page faults	273	660	1230
Instructions	28,974,520	35,669,663	2,050,000,720
Time	0.028951596	0.061482514	0.540514533
200KB			
Page faults	326	713	1,334
Instructions	45,926,310	52,487,839	2,414,208,538
Time	0.063752567	0.076081896	0.66811976
500KB			
Page faults	525	912	1,734
Instructions	111,338,010	119,492,290	4,070,225,945
Time	0.117933531	0.127791843	1.201331268

TABLE I
PERFORMANCE EVALUATION OF OUR LANGSEC VALIDATION FILTERS USING `PERF`. TO BENCHMARK THE OVERHEAD IMPOSED BY OUR LANGSEC FILTERS, WE SELECTED PNG FILES OF VARYING SIZES AND RAN THEM THROUGH A BARE-METAL EXECUTABLE, AND ONE WITH THE LANGSEC FILTER, AND ONE WITH AN ELFBAC POLICY TO ENSURE CFI.

We also demonstrate that constructing a single parser for an input format using a hardened parser combinator library we can effectively prevent *multiple* exploits of the same kind. To be more exact, a LangSec filter for a given grammar will reject all syntactically invalid crafted input exploits.

This is especially beneficial for software security because a single correctly written LangSec patch for a given vulnerability can successfully prevent multiple exploits discovered in the future. As we show in our example both CVE-2004-059,[3] and CVE-2010-1205[4] are prevented using the same filter thus eliminating the need to patch the software multiple times for different exploits of the same kind.

B. Performance

To evaluate the implementation, we ran the `perf` linux profiler on three different implementations. The first implementation was run without any instrumentation, the second with the filter but without using ELFbac, and the third with the filter and a sufficient ELFbac policy. We ran the experiments for PNG files of varying sizes. The results are shown in Table I.

The ELFbac policy we implemented for the implementation is in 3. This policy has four states. Each state restricts the code and data sections the application is allowed to access. The four states are for `libc`, the LangSec filter, `libpng` library, and the starting state. ELFbac not only restricts the memory these states are allowed to access, but also only allows a transition between them when the state executes a function defined in the policy (viz. you can only jump from specific locations in the code). In the example shown, the starting state `main`, is allowed to only transition to the LangSec filter or the `libc` state. Thus, preventing any jumps directly into the vulnerable library.

As shown in the table, we profiled the effect of the PNG LangSec filter with PNG files of varying sizes. We observed that the performance overhead increases with the size of the input, as the LangSec parser has to validate it before it is consumed by the third party library. We observe that the overhead is on average less than 25% when using the LangSec filter.

[3]https://www.exploit-db.com/exploits/14422
[4]https://www.exploit-db.com/exploits/389

The overhead, however, increases when using CFI techniques such as ELFbac. ELFbac forces page faults when memory is fixed accessed in a particular state to ensure that the program has access to the particular memory location. ELFbac also triggers a TLB flush on every state transition. These are large performance costs. The ARM64 version of ELFbac does not incur such high performance costs because of the way it handles page faults and TLB flushes, and can be used to reduce overheads.

V. RELATED WORK

The defenses we presented in this paper serve two purposes: defending against weird machines using LangSec filters, and ensuring that the filters cannot be bypassed. Memory safety techniques solve some of the same problems as our LangSec filters, whereas control-flow integrity techniques ensure that attackers cannot bypass the filters.

There are several techniques that address memory safety [16], [17] and control-flow integrity. In this section, we summarize memory safety techniques that address the same problem as our LangSec filters, and control-flow defense techniques that can ensure that our LangSec filters are not bypassed.

A. Memory Safety Approaches

Attackers use memory corruption in software to construct exploits. Memory safety exploit-mitigation techniques watch for memory safety violations in a program and terminate the program upon detection, thereby preventing the exploit.

Memory corruption can be spatial or temporal, or both. Spatial memory corruption occurs when a pointer accesses an address beyond the intended memory region. Whereas, temporal memory corruption occurs when an object is accessed before creation or after it has been deallocated. For example, a use-after-free is a temporal memory safety violation. On the other hand, a buffer overflow is a spatial memory safety violation.

1) Spatial Memory Safety: Several recent papers have added run-time checks on pointers to enforce spatial memory safety in software. In *Cyclone*, a dialect of C, the programming can control the memory layout and keep track of the address ranges being used [18]. This information is stored by altering the pointer representation to include the boundary information within it. These modified pointers are referred to as "fat pointers." We see two issues with Cyclone. First, changing the pointer representation causes substantial binary incompatibility issues in the resulting binaries. Second, Cyclone requires source code annotations to guide the static analysis which makes it impractical for large existing unannotated code bases.

SoftBound [19] addresses the binary incompatibility issues caused by Cyclone, by separating the boundary information from the pointer representation and keeping this information in a data structure elsewhere. Softbound is implemented as a compile-time transformation for C programs. Code is instrumented to update the data structure which tracks the boundary metadata when necessary. Softbound addresses spatial memory safety concerns only and has a 67% performance overhead.

2) Temporal Memory Safety: A shortcoming of object-based bounds checking is that they are unable to detect memory corruption *within* objects and thus may lead to false negatives. An undetected memory corruption within an object could potentially lead to an exploit.

Valgrind's Memcheck [20] and AddressSanitizer [21] detect use-after-free bugs by keeping track of the allocation and deallocation information in a separate memory region. This allows them to detect dereferences of dangling pointers. If, however, a memory region is reallocated for some other object then neither Memcheck nor AddressSanitizer can detect an invalid access in that memory region.

The CETS project [22] extended SoftBound [19] by adding temporal pointer validation, thus ensuring complete memory safety. CETS like SoftBound is also a compile time transformation. CETS uses a dictionary to keep track of the validity of every object allocated by the program. If an object is invalidated, its associated entry in the dictionary is updated to reflect the change. On every pointer dereference, CETS checks if the pointer is valid before allowing the access. Code instrumented by CETS is also formally proven to ensure temporal memory safety provided the code does not have any spatial memory safety violations.

DANGNULL [23] prevents temporal memory safety violations, by setting all pointers to `NULL` when an object is freed. `DANGNULL` requires static instrumentation done via an LLVM IR pass. This instrumentation looks for pointer assignments and inserts a trace function. Using this generated trace the DANGNULL run-time library detects and converts all dangling pointers to NULL. DangSan [24] and FreeSentry [25] also invalidate the pointers pointing to an object when the object is freed.

Undangle [26] is a tool that detects dangling pointers via taint analysis of a binary. It does not require access to source code, however if source code is available it can use it to augment its analysis. Since, Undangle works on the execution trace of a program, it does not impact the performance directly. Undangle, is different in the sense that it focuses on early detection of temporal memory safety violations, instead of prevention at run-time.

The LangSec filters we use in our paper are different from all these memory-safety techniques. We formalize the data format and implement parsers for them. LangSec filters require developers to formalize the input beforehand. This leads to better software, by design.

B. Control Flow Integrity

Control Flow Integrity (CFI) techniques prevent exploits from altering the control flow of the program. CFI techniques deter exploits by detecting an anomalous program execution path and halting the program when it occurs thereby mitigating the exploit. They do so by validating the memory locations holding the control flow information of the program. In *Armor Within*, we make use of ELFbac to ensure control-flow integrity.

Stack cookies [4], [27] prevent the corruption of the return address by placing a random value among the contents of the stack and verifying it's integrity before using the return address stored on the stack. This prevents buffer overflow attacks where the stack is corrupted by writing beyond the memory allocated for the a stack object. It is important to note that an attacker can bypass the stack cookie protection if they can directly overwrite the return address.

Abadi et al. [7] stipulated that if a program instruction affects the control flow of the program, it should do so in accordance to the "intended" program execution. The CFI technique designed by Abadi et al. [7] inserted checks in the program binary to ensure machine instructions transferring control adhere to the extracted control flow graph (CFG) of the program. A unique ID is assigned to each destination address that any program instruction may transfer it's control to. A check is added in the program machine code to ensure when an instruction transfers control, the destination has the correct ID.

A major drawback of CFI is that instrumented and uninstrumented software modules are not compatible with each other. Furthermore, in the absence of a fine-grained CFG, the CFI policy does not offer sufficient protections. Since the control-flow policy cannot distinguish between targets within an equivalence class, if a policy is coarse grained an adversary may be able to "bend" the policy without being detected [28]. In this paper, we used ELFbac and Clang's CFI implementations to ensure control-flow integrity. ELFbac suffers from the drawback we mentioned above—it requires a fine-grained policy.

VI. Conclusion

In this paper, we presented *Armor Within*, comprising two techniques to inject LangSec parsers in binaries to protect the binary against insecure libraries. In the first technique, we rewrote the symbol table of a binary to force it to run through the LangSec parser before the control flow enters the library. In the second technique, we lifted binaries to LLVM IR and injected the parsers in the IR. We also added the metadata needed for the binaries to work with ELFbac. We demonstrated the effectiveness of our techniques on two popular exploits on `libpng` and `libxml`. We found that our tools were fast and effective, and added minimal overhead to existing binaries.

We are aware that rewriting the Procedure Linkage Table (PLT) of an ELF binary is an existing less invasive binary rewriting approach. The PLT entry for the vulnerable function of the module could be rewritten to jump to the code for the LangSec filter. The code for the LangSec filter could be added as an additional executable segment in the ELF binary. However, we did not pursue this method as it can only insert the LangSec filter to vulnerable functions which are part of a dynamically linked library.

In future work, we are taking various new directions. We are systematizing exploits and exploit mitigation techniques to formalize which exploits can be mitigated with *Armor Within*. Although we used Hammer [9] in this paper to build our LangSec parsers, we can use other libraries. We are building a formally verified parser-combinator toolkit to ensure that the toolkit is bug-free.

VII. Acknowledgements

We thank the anonymous reviews for their comments and suggestions which greatly improved the manuscript.

References

[1] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to "weird machines" and theory of computation," *USENIX; login*, vol. 36, no. 6, 2011.

[2] MITRE, "CVE-2004-0597 : Multiple buffer overflows in libpng 1.2.5," Available from MITRE CVE-ID CVE-2004-0597, 2004.

[3] A. van de Ven and I. Molnar, "Exec shield," 2004. [Online]. Available: http://www.redhat.com/f/pdf/rhel/WHP0006USExecshield.pdf

[4] H. Etoh and K. Yoda, "Memory device, stack protection system, computer system, compiler, stack protection method, storage medium and program transmission apparatus," Sep. 6 2005, uS Patent 6,941,473.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*. San Francisco, CA: USENIX, 2011, pp. 25–41.

[6] Microsoft, "/SAFESEH (Safe Exception Handlers)," 2003, [Online; accessed 21-September-2019]. [Online]. Available: http://msdn2.microsoft.com/en-us/library/9a89h429.aspx

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information Systems Security*, vol. 13, no. 1, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[8] PaX, "Address Space Layout Randomization," 2001, [Online; accessed 21-September-2019]. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[9] M. Patterson, "Parser combinators for binary formats, in C," 2015, [Online; accessed 4-March-2020]. [Online]. Available: https://gitlab.special-circumstanc.es/hammer/hammer

[10] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1465–1482. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[11] G. Couprie, "Nom, a byte oriented, streaming, zero copy, parser combinators library in rust," in *IEEE Security and Privacy Workshops*. San Jose, CA: IEEE, 2015, pp. 142–148.

[12] J. Bangert, S. Bratus, R. Shapiro, J. Reeves, S. W. Smith, A. Shubina, M. Koo, and M. E. Locasto, "Sections are types, linking is policy: Using the loader format for expressing programmer intent," *BlackHat USA*, 2016.

[13] J. Bangert, S. Bratus, R. Shapiro, M. E. Locasto, J. Reeves, S. W. Smith, and A. Shubina, "ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection," Dartmouth College, Computer Science, Hanover, NH, Tech. Rep. TR2013-727, May 2013. [Online]. Available: http://www.cs.dartmouth.edu/reports/TR2013-727.pdf

[14] P. Anantharaman, K. Palani, R. Brantley, G. Brown, S. Bratus, and S. W. Smith, "Phasorsec: Protocol security filters for wide area measurement systems," in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. Aalborg, Denmark: IEEE, Oct 2018, pp. 1–6.

[15] Trail of Bits, "mcsema," https://github.com/trailofbits/mcsema, 2018.

[16] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy*. San Francisco, CA: IEEE, May 2013, pp. 48–62.

[17] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," in *IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, May 2019, pp. 1275–1295.

[18] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*. Monterey, CA: USENIX, 2002, pp. 275–288.

[19] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[20] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," *ACM Sigplan Notices*, vol. 45, no. 8, pp. 31–40, 2010.

[23] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in *NDSS*. San Diego, CA: Internet Society, 2015.

[24] E. van der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 405419. [Online]. Available: https://doi.org/10.1145/3064176.3064211

[25] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers," in *NDSS*. San Diego, CA: Internet Society, 2015.

[26] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the International Symposium on Software Testing and Analysis*. Minneapolis, MN: ACM, 2012, pp. 133–143.

[27] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554

[28] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini