23

Intent as a Secure Design Primitive

Prashant Anantharaman[1], J. Peter Brady[1], Ira Ray Jenkins[1], Vijay H. Kothari[1], Michael C. Millian[1], Kartik Palani[2], Kirti V. Rathore[3], Jason Reeves[4], Rebecca Shapiro[5], Syed H. Tanveer[1], Sergey Bratus[1], and Sean W. Smith[1]

[1] Department of Computer Science, Dartmouth College, Hanover, NH, USA
[2] Dartmouth College, Hanover, NH, USA
[3] Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Champaign, IL, USA
[4] VMWare, Inc., Palo Alto, CA, USA
[5] Champlain College, Burlington, VT, USA

23.1 Introduction

At the core of design lies *intent*, or the primary goals and objectives that drive the creation and usage of a technology. However, there are often multiple actors involved with a technology's production, all with their own ideas about intent.

- The designer has some idea of what their product should do and how it should do it.
- The developer produces code to realize their own interpretation of the designer's intent.
- The user uses the product according to their own intent, which may or may not match those of the designer and developer.

This chapter examines the role of intent when it comes to *security*. The security of a technology stems from how closely the intents of its designers, developers, and users align, as well as whether these intents reflect reality. However, ensuring that

Modeling and Design of Secure Internet of Things, First Edition. Edited by Charles A. Kamhoua, Laurent L. Njilla, Alexander Kott and Sachin Shetty.

these intents align and match reality can be difficult or even impossible. Designers tend to explicitly define the functionality of their technologies, but their security intent is often implicit and conceptualized in vague terms like "only legitimate users will be able to authenticate." This oversight is not intentional; rather, the designer lacks the proper tools to think through the security implications of their design decisions. Developers must then translate these vague notions into code, and their decisions may not align with the designer's mental model. Finally, users have their own goals and intentions; their focus is often on accomplishing their primary task rather than ensuring the system remains secure. As they are the final arbiters of how the system operates, their decisions have the most impact on the system's security, and their usability-centered choices can override the designer's and developer's security intentions.

Intent mismatches have caused many of the most severe and prevalent vulnerabilities, and this trend is continuing in the Internet of Things (IoT) era. The Mirai botnet [1], which utilized buffer overflows to infect routers, demonstrates one such vulnerability. Inadequate input validation on routers violated the designer's assumption that only properly formatted input would be accepted. Another example involves the first variant of the Spectre vulnerability [2], which demonstrates a mismatch between the intents of the processor designer and the operating system developer. Speculative execution of program instructions violated the developer's assumption that certain code branches would never be executed and should therefore leave no trace in the system hardware. Unchanged default passwords are a mismatch between developer and user intent; developers intend for users to create secure passwords once they start using the system, whereas users may not do so as they are reluctant to change anything once the system is running – and in some application scenarios, users even depend on well-known default passwords for system availability during crisis situations.

The unique nature of the IoT exacerbates the security ramifications of these intent mismatches. Almost by definition, the IoT brings computing to physical-world devices that have historically lacked such computational capabilities, thus merging previously separate design goals. Inexperienced designers who lack adequate security background or training may be thrust into the role of bringing their revolutionary vision to fruition. A lack of standards and conventions means many developers will develop ad-hoc protocols or put together makeshift devices that satisfy the demands of their use case without giving security adequate consideration. The unprecedented scale of the IoT presents an additional challenge. For example, when users do not change their default passwords, millions of unsecure devices could be compromised to create a massive botnet. These challenges render traditional security assumptions and models obsolete, requiring us to create new approaches suitable to the current landscape.

In this chapter, we present two emerging security paradigms we have helped develop to preserve designer and developer intent.[1] The growing field of *language-theoretic security* (*LangSec*) provides a theoretical foundation, as well as the machinery, to ensure that devices only accept developer-intended inputs, thus protecting these devices against a myriad of input-validation bugs. LangSec posits that input recognizers must be built from the formal specification of a protocol, and the recognizer must adopt a language-based approach to accept valid input while rejecting all invalid input without performing additional computation. LangSec compels the developer to contemplate the input language of the protocol being designed and implemented. Separately, *ELF-based access control* (*ELFbac*) empowers the developer to specify access control policies for their programs at a natural level of granularity, the *application binary interface* (*ABI*). ELFbac allows the developer to codify their intent by creating policies for intra-process memory isolation. Together, these techniques help to reduce the incidence (the purview of LangSec) and consequences (the purview of ELFbac) of zero-days, which are more critical in the IoT where patching is harder and the consequences of compromise may be so wide reaching. By restricting the input a program can accept and the memory it can access at any given time, these techniques effectively constrain and/or prevent undesirable behavior, forcing the program to conform to the intent of the developer (see Figure 23.1).

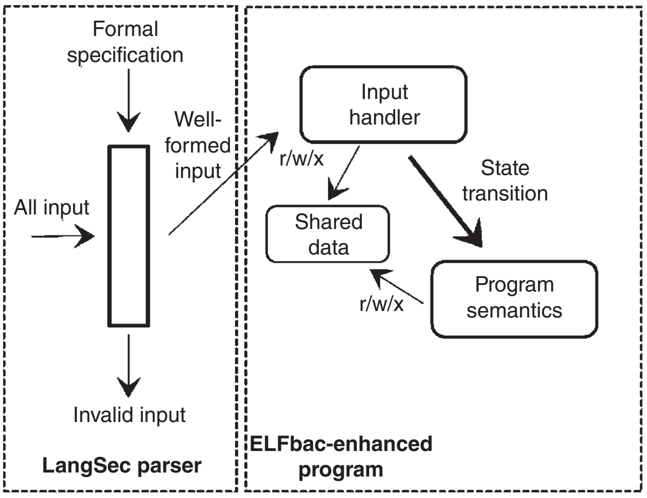


Figure 23.1 ELFbac and LangSec both enforce human intent by codifying the specifications.

1 Preserving user intent is also important, but it is not a focus of this chapter.

Section 23.2 presents LangSec. Section 23.3 presents ELFbac. Section 23.4 presents an IoT application area where we use the two techniques together. Section 23.5 discusses how we evaluate implementations, and Section 23.6 concludes.

23.2 A LangSec Primer

The field of *language-theoretic security* (*LangSec*) posits that since exploits are input-driven computations which are unintended by and unknown to designers and developers, defending against exploits requires taking a principled approach to input recognition that is rooted in *formal language theory* and *computability theory*. (Security based on *programming languages* is an orthogonal topic.)

LangSec examines both theoretic and applied challenges with input sanitization and input recognition that manifest in real-world exploits; it suggests parser design and implementation best practices, and it even provides a tool that facilitates the principled development of parsers – all with the goal of preserving designer and developer security intent.[2]

23.2.1 What Is LangSec?

The set of acceptable inputs to a system is often not explicitly defined by the developer, nor is it tested extensively. This leads to invalid input being processed instead of being rejected, which in turn leads to major bugs such as Heartbleed [3], Shellshock [4], and Android Master Key [5]. When a system receives an unanticipated input, the processing code may drive the system to a state that is unaccounted for by the developer. LangSec seeks to prevent such vulnerabilities driven by poor input handling.

At its core, LangSec is the idea that input-driven security exploits can be avoided by ensuring that the acceptable inputs to a program:

- are well-defined by a language grammar,
- are as simple as possible within the Chomsky hierarchy, and
- can be fully validated by a dedicated parser of appropriate power as defined by the Chomsky hierarchy.

The idea with LangSec-hardened parsers is to separate the input validation code of the program from the rest of the code, so that the main program never acts on input that has not been validated. The set of acceptable inputs is used to define a

2 LangSec offers many other contributions as well, such as identifying anti-patterns. Our focus in this section, however, is to give the reader an overview of the field.

grammar for a language. A LangSec parser must simply follow this grammar by rejecting any non-conforming input and operating correctly on well-formed input.

23.2.2 Exploits Shatter Intent

The designer and the developer often have some notion of what they want their software to do and not to do. Correct behavior may be explicitly expressed at different points within the design and development process – and it may even be enforced in the final product. However, it may also be an amorphous, ill-defined, high-level notion that is assumed, rarely thought about, and never explicitly stated. Many times, we see something in between, where attempts to express intent are occasionally made throughout the design and development process, but this intent cannot be correctly enforced because the designer and developer lack the requisite knowledge or tools to ensure the security principles they want are achieved in the software they produce. This is perhaps best exemplified by the anti-pattern of the shotgun parser [6], which comprises segments of code scattered throughout a program that collectively aim to sanitize the input but instead exhibit security vulnerabilities because input-driven computation occurs before the input is deemed legitimate or because the parser code is executed at different times and may not act on a single input as the developer expects. These shotgun parsers demonstrate that their developer not only intends to protect against bad input, but also that they expend great effort in pursuing this endeavor; however, despite this effort, shotgun parsers frequently fail to safeguard the program from malicious-input-driven exploits, as evidenced by the many exploits seen in the wild that leverage unintended computation enabled by these poorly designed parsers.

Indeed, exploits shatter designer and developer intent – and they do so by design. A computer exploit is input that is crafted and submitted to a target program to produce input-driven computation unintended by and unbeknownst to the designer and the developer [7]. First, the exploit programmer seeks to uncover unintended computational capabilities offered by the target program. Next, the exploit programmer identifies and distills these behaviors by determining simple input constructs that reliably produce them. Finally, the exploit programmer chains together these input constructs to create the exploit.

We stress that programming an exploit is programming. The basic unit or building block of an exploit is the "weird instruction," a gadget or a collection of sequentially executed instructions that collectively performs a useful operation for the exploit programmer, one that should not be allowed as neither the developer nor designer intended for such computational capabilities. Once the exploit programmer discovers and distills unintended behaviors into these weird instructions, they systematically piece these weird instructions together to create the exploit, just as, say, an assembly programmer pieces together assembly instructions to create an assembly program. The exploit runs on – and therefore serves as an attestation to the existence of – the "weird machine," a programmable machine harbored by the target program that offers the requisite weird instructions to construct the exploit.[3]

The exploit will not exist if the weird machine upon which it runs does not exist. LangSec attempts to prevent the emergence of weird machines.

23.2.3 A Brief Detour into the Theory of Computation

LangSec builds upon the Theory of Computation. Here, we give a very brief primer of the theory of computation and its branches to provide the requisite language and machinery to understand the theoretical underpinnings of modern-day exploits. For the reader who seeks a more complete treatment, we highly recommend Sipser's book [8].

In formal language theory, an *alphabet* is a non-empty finite set of *symbols*. A *string* over an alphabet is a finite sequence of symbols belonging to that alphabet. For example, consider what is colloquially considered the English alphabet: $\Sigma = \{a, b, \ldots, z\}$. Here, $a \in \Sigma$ is a symbol belonging to the alphabet and the word $cat \in \Sigma^*$ is a string over the alphabet. A *language* is defined in relation to an alphabet as a set of strings over that alphabet. A natural way to express a language is to give a grammar. A *grammar* specifies rules, each of which is a mapping from a variable to a sequence of variables and symbols. A grammar comprises a start symbol along with a sequence of rules. The language of the grammar is simply the language comprising all strings generated by the start symbol. These language-theoretic notions form the building blocks of automata theory and computability theory – the connection being what computation is required to "capture" languages. These fields, in turn, lay the foundation for language-theoretic security.

As suggested by its name, the central focus of automata theory is *automata* – or mathematical models of computation – and their capabilities. Core to automata is the notion of *state*, which roughly refers to a condition that may dictate a subsequent course of action. Automata maintain state, often have some form of memory, and perform computation on input in its pursuit of some task, such as generating output or determining whether the supplied input is well-formed.[4] The task that we are concerned with in this chapter, which is perhaps the most common task, is to accept or reject input. The automaton begins in a start state. The automaton (or perhaps its operator) then repeatedly examines the state that the automaton is in. Based on that state, it will read symbols from the input and/or the memory that is part of the automaton, such as a stack or a tape, it may

3 For further discussion of exploit programming, weird instructions, and weird machines, we recommend reading previous work by Bratus et al. [7].

4 The standard concepts of Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA) are just special cases of these machines.

perform an action such as writing a symbol to memory, and it will transition to the next state. When supplied with input, the automaton may either run indefinitely or it may terminate when some condition is met, such as there being no symbols left to process. If the computation terminates, the state that the automaton is in during termination determines whether the input is to be accepted or rejected. That is, the automata we are interested in take as input a string and either *accept* or *reject* the string. An automaton *recognizes* a language if the automaton accepts only those input strings that belong to that language. And it *decides* a language if it accepts those strings that belong to the language and rejects those that do not (instead of sometimes merely running forever).

Central to discussion of languages, grammars, and automata is the notion of *expressiveness*, which enables us to meaningfully differentiate classes of a given type. A language L is considered more *expressive* than a language L' if L is a strict superset of L'. Similarly, a class of languages $\mathcal{L}$ is more expressive than a class of language $\mathcal{L}'$ if $\mathcal{L}$ is a strict superset of $\mathcal{L}'$. Similar notions of expressiveness exist for classes of grammars and automata, grounded in the languages associated with these grammars and automata. Indeed, this notion of expressiveness intimately links classes of languages, grammars, and automata, e.g. the grammars corresponding to finite state automata are regular grammars and the corresponding languages are regular languages. Moreover, it enables us to develop nested classification schemes such as the well-known Chomsky hierarchy. We find, for example, that in the Chomsky hierarchy, finite state machines (and the corresponding grammar and language classes of regular grammars and regular languages) are much less expressive than Turing machines (and the corresponding grammar and language classes of unrestricted grammars and recursively enumerable languages).

Computability theory and computational complexity theory provide us with a foundation for understanding the limitations of automata in regard to what they can do and how efficiently they can do it. Computability theory, in particular, provides us valuable negative results that inform LangSec.

23.2.3.1 The Halting Problem, Undecidability, and the Equivalence Problems

A particularly well-known, expressive class of automata is Turing machines. Certain classes of languages are not recognizable using a Turing machine and there are still more that are undecidable. The classic example of a Turing-undecidable language comes from the Halting Problem, which involves designing a Turing machine that accepts all (M, I_M) pairs where M is a Turing machine and I_M is an input for which M halts when run on I_M and rejects all other Turing machine, input pairs.

It turns out that while the language of the halting problem, L_{TM}, is not decidable, it is recognizable. However, Turing-unrecognizable languages also exist; one such example is the complement of L_{TM} [8]. Recursive languages are exactly those languages that are decidable; notably, the slightly less expressive context-sensitive languages in the Chomsky hierarchy are also decidable.

Undecidability is relevant to LangSec because if an input language is computationally undecidable, then it *cannot* be validated – a validation layer will always let some crafted attack input through. It is also relevant because of the *equivalence problem* – determining whether the language of one grammar is equivalent to the language of another. In terms of LangSec: Do two input validation layers accept the same input? While this problem is undecidable in the general case and for many context-free languages, it is decidable for deterministic context-free languages [9].

23.2.3.2 An Extension to the Chomsky Hierarchy

The well-known Chomsky hierarchy provides a containment classification for grammars, as well as their corresponding languages and automata, based on their expressiveness. With LangSec, we are largely concerned with the questions of decidability that we presented earlier. Hence, we extend the Chomsky hierarchy in Figure 23.2 by differentiating between non-deterministic pushdown automata and deterministic pushdown automata, where a crucial barrier lies with regard to the decidability of parser/grammar equivalence. We also note the barrier of decidability at linear-bounded automata within the hierarchy.

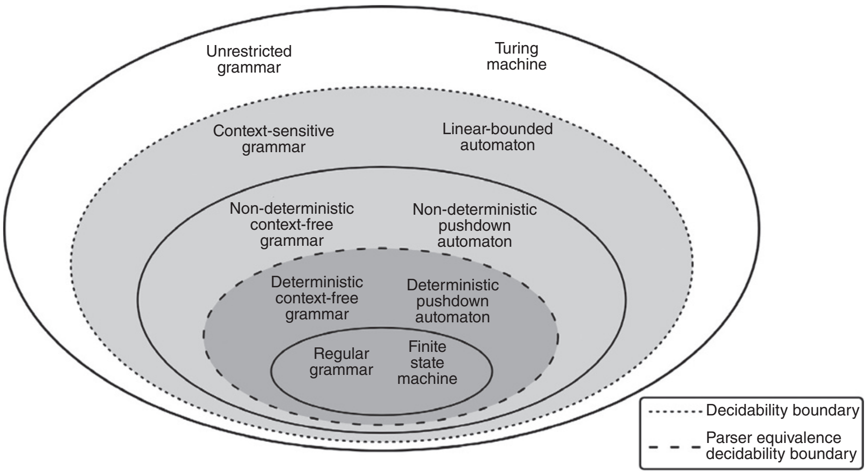

Unrestricted grammar
Turing machine
Context-sensitive grammar
Linear-bounded automaton
Non-deterministic context-free grammar
Non-deterministic pushdown automaton
Deterministic context-free grammar
Deterministic pushdown automaton
Regular grammar
Finite state machine
Decidability boundary
Parser equivalence decidability boundary

Figure 23.2 Chomsky hierarchy extended with LangSec boundaries.

23.2.4 Input Recognition

Sassaman et al. [7] state that "a system's security is largely defined by what computations can and cannot occur in it under all possible inputs." A program's input drives its control and data flow, and input-driven computation that does not conform to the intent of the designer and developer suggests the existence of a weird machine that an adversary can operationalize to program an exploit. This raises the question: How do we produce programs that match the intent of the designer and developer with the end goal of preventing such exploitation? LangSec, in its pursuit of an answer to this very question, advocates treating the input as a formal language, making this language as simple as possible, and ensuring that the input is correctly recognized in accordance with the design specification before permitting input-driven computation.

Ensuring the absence of unintended input-driven computation is intimately linked with the objective of secure composition. Sassaman et al. [7] argue that, as composition is integral to the construction of complex systems, secure composition has become a colossal security challenge. Modern systems comprise many components that perform a variety of tasks across different layers, and these components must be able to talk to one another. The challenge of secure composition lies not only in securing each of the component parts, but also securing the interfaces, i.e. the glue that enables communication between these parts. The vulnerabilities underpinning modern-day exploits often stem from improper input handling at these interfaces.

To secure programs and the interfaces between them, it is imperative that we harden the *parser*, the code responsible for input handling. A primary goal of input handling is to – or, rather, if we are to produce secure code, it *should* be to – *recognize* the input language. That is, we want an input handling routine, a *parser*, that only accepts strings that belong to the input language. However, this alone is insufficient; ensuring recognizability makes no guarantees about whether the parser will halt on all inputs – and we truly want our parsers to halt on all inputs! Ergo, what we really want is for the parser to *decide* the input language, that is, to accept strings that belong to the input language and to reject strings that do not. We may wish for additional properties from the input language and the parser, e.g. we may want to satisfy timing constraints, space constraints, or other objectives, but at a bare minimum, the input language must be decidable.

23.2.4.1 Full Recognition Before Processing

LangSec advocates recognizing the input language – or more precisely, deciding the input language – before the program performs any (non-parser) computation on that input. That is, the parser should be separated from the rest of the program and the parser should reject all invalid input immediately, only allowing valid input to be passed on to the program internals. This approach is expressed in Figure 23.3.

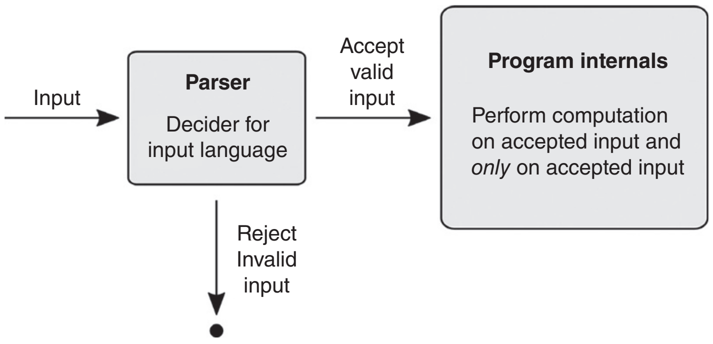

Input
Parser
Decider for input language
Accept valid input
Program internals
Perform computation on accepted input and *only* on accepted input
Reject Invalid input

Figure 23.3 Full recognition before computation.

23.2.4.2 Principle of Least Expressiveness

The *principle of least privilege* roughly states that an entity within a system, such as a user or a process, should operate with the least privilege required to perform the task at hand [10]. LangSec advocates a similar principle for the design of protocol specifications or grammars and their parser implementations, the *principle of least expressiveness*:

> *One should always use the least expressive grammar, language, or automaton that achieves the task.*

This principle has the following two implications:

- Protocols should be designed to be minimally expressive.
- Parsers should be no more expressive than is required to decide the languages specified by the protocols they obey.

23.2.4.3 Principle of Parser Computational Equivalence

As mentioned earlier, secure composition is one of the leading security challenges. As such, it's vital to the security of interfaces. Given two parsers that act on the same input, those parsers must wholly agree, i.e. for every possible input, either they both accept the input or they both reject it. The *principle of parser equivalence* states:

> *Secure composition requires any two parsers that should decide the same language, e.g., due to sharing a component–component boundary, do decide that language.*

Recall that the equivalence problem is not decidable for non-deterministic context-free languages and more expressive languages, whereas it is decidable for deterministic context-free languages. When constructing parsers involved in secure composition, one should, therefore, aim to ensure they are no more expressive than deterministic context-free.

23.2.4.4 Updating Postel's Robustness Principle

Jon Postel's robustness principle states, in the context of TCP implementation: "[B]e conservative in what you do, be liberal in what you accept from others." [11]. While this may be sage advice in many situations, its misinterpretation has led to many Internet bugs. In particular, the advice does not account for the active adversary. While the principle, when read in context, displays understanding of bugs caused by weak input handling and also states that such a principle must be applied at every layer of the network stack, protocol implementers and the implementations, by extension – often mistakenly assume that the input that one layer passes to another is well-formed and any malformed input is filtered out at the layer boundary.

Previous work by Sassaman et al. [12] proposes an update to the robustness principle, which is particularly pertinent to the new network protocols being developed for the IoT:

- Be definite in what you accept. Clearly stating what language your machine understands is critical to the security of the device.
- Treat all input as a language that is parsed by an automaton of matching computational power. Also, the recognizer must be generated from the grammar of the language. Try to keep the language regular or at most context-free.

23.2.5 Incorporating LangSec into System Design and Development

Given the importance of specifying an input language formally at the protocol design phase, we want to make it as easy as possible for the parser developers – those people writing production code – to translate the language specification correctly. This job involves taking a grammar – the description of a language as a state machine or in prose – and writing code to implement the grammar. In the traditional workflow, every parser is written by hand using standard code blocks. In practice, it is difficult to determine that a series of *if* statements corresponds to the intended grammar. Any misunderstanding or oversight in reading the grammar on the part of the developer may result in an implementation that does not match the specification. We see this kind of mistake happen in many real-world parser bugs like Heartbleed [3] and Android Master Key [5]. In a formal sense, even a small difference changes the implementation to one for a completely different language. Additionally, writing a parser as a series of code blocks involves performing the entire translation process from scratch every time.

A LangSec solution to this problem is to use a *parser combinator tool*. The building blocks of grammars are combinators like concatenation, union, and Kleene star. A parser combinator tool is just a library that provides these combinators. In this way, the developer ends up with code that visually looks like the grammar. Verifying that the implementation matches the specification is trivial and can be done by inspection. Furthermore, the correctness of each of the combinators can be verified independently, meaning the developer does not have to worry about translating the combinators into code correctly – only calling the combinators as the grammar specifies.

23.2.5.1 Hammer

Hammer is a parser combinator tool written in C with bindings for several other languages, including C++, Java, Python, Ruby, Pearl, PHP, and .NET. Hammer provides implementations of all necessary combinators for context-free grammars and then some!

Future directions include adding a utility to create fuzz data based on the parser implementation and adding a utility to take a specification such as a state machine as input and automatically generate a parser (see Table 23.1).

Table 23.1 Syntax and usage of Hammer.

Syntax	Usage	Semantics
`h.ch`	`h.ch('a')`	Matches a single specified character token.
`h.ch_range`	`h.ch_range('a','z')`	Matches a single token in the specified character range.
`h.uint8`	`h.uint8()`	Matches a single integer token.
`h.int_range`	`h.int_range(1,16)`	Matches a single integer token in the specified range.
`h.sequence`	`h.sequence(h.ch('a'), h.ch('a'))`	Performs the concatenation operation.
`h.choice`	`h.choice(h.ch('a'), h.ch('b'))`	Performs the Boolean "or" operation.
`h.many`	`h.many(h.ch('a')`	Performs the Kleene Star operation.
`h.optional`	`h.optional(h.ch('a'))`	Specifies that matching this token is optional.

23.2.6 Summary

LangSec provides a theoretical foundation, a body of research, and usable tools for the design and implementation of protocols and parsers to avert many of the worst exploits of the modern day. The IoT and the incentive structure that drive both fledgling and established companies to rush to get broad market coverage have resulted in more component–component interactions with less focus on secure input handling. LangSec provides the fix: separate the parser from the remainder of the program and ensure it matches the specification; ensure parsing is done in full by the parser and only pass on valid input to the program internals; use the least expressive computation power necessary; and ensure parser equivalence. LangSec also delivers a tool in Hammer to facilitate development of parsers that match their underlying specifications.

23.3 An ELFbac Primer

In this section, we discuss *ELF-based access control* (*ELFbac*), a technique for intra-process memory isolation that can be used to mediate what code can operate on which data at what times. By allowing the user to express their intent through policy, ELFbac can protect programs from intent-violating exploits. We provide a tutorial on designing and enforcing ELFbac policies, and we show how ELFbac can mitigate existing high-profile vulnerabilities such as the SSH roaming bug and variant 1 of the Spectre processor bug.

23.3.1 What Is ELFbac?

The *principle of least privilege* states that the components of a system should have the most restrictive set of permissions to accomplish their tasks [10]. This level of isolation limits the exposure of vulnerabilities within a system. For example, a web browser should not have default access to critical operating system files. In this way, vulnerabilities in one component do not impact the overall operation of the system.

With ELFbac, we apply this principle all the way down to the individual code and data elements within a single process address space. A bug in one library, the browser uses could trigger functions from other libraries.

Privilege separation is critical to preserving developer intent. However, existing access control mechanisms do not address such intent. File system, process, or system call level policies are not granular enough for processes which may keep their objects in memory and never trigger an offending operation until it is too late. Mitigations for memory corruption vulnerabilities such as control flow integrity operate at finer granularities but do not address a "Trojan horse" attack where a malicious third-party library mapped to the process address space can read and manipulate process memory it should not have access to.

To this end, ELFbac preserves developer intent within the address space. We frame the problem of unintended, intra-process memory accesses involving buffer overflows or use-after-free as an out-of-type reference problem. We provide a technique for constructing a type-state system consistent with the existing ABI to allow developers to express intent.

23.3.2 Why ELFbac?

While several similar memory protection schemes have been proposed in the past, they do not achieve the granularity of protection that we desire here:

- Standard sandboxing is too coarse-grained for our purposes, as it requires us to "spawn extra processes or re-engineer code" [13] to achieve the code/data-level separation we need.
- SELinux [14] extends traditional Unix file protections to define how programs can interact with other objects, but cannot "protect a process' data from the process itself" [13].
- Mondrian [15] allows users to set the read/write/execute permissions to individual memory segments at the word level, but doing so required additional hardware mechanisms (more registers, a "permissions lookaside buffer") that are not currently present in commodity systems.[5]

23.3.3 Relationships Between Code and Data

A process has code chunks and data chunks. Standard OS controls provide little protection on what they can do to each other – but the developer certainly has intended behavior. We would like to have the program development process automatically give us info about what these chunks are and what the developer intends.

Systems already have standard ways to have an "executable" program file express the code and data for a given executable, as well as the metadata necessary for the creation of a process address space. In our work, we focus on the *Executable and Linkable Format* (*ELF*), standard in the UNIX family (although our approach can easily generalize to other formats). Inherent within the ELF format are specific rules and conventions that are expected to be followed – for example, code in the `.init` section operates only on data in the `.ctors` section. ELFbac's key insight is that ELF sections can be leveraged to encode *explicit* rules beyond standard ELF

5 Later papers used the Dutch spelling *Mondriaan* for this project.

conventions, allowing users to create new sections that contain (i) specific pieces of code or data, and (ii) policies that govern the interaction between other sections [13].

To enforce these permissions, however, we must overcome the issue of the “forgetful loader” [13]. When a program is loaded into memory to be run, its layout is organized by *segments* rather than sections. Segments are grouped together based solely on their read/write/execute permissions, causing section-level permissions to be discarded and leaving the door open for programmer intent to be ignored. To get around the forgetful loader, we need a way to maintain section-level permissions all the way through program execution.

23.3.4 Design

ELFbac utilizes the existing linking and loading process to define policies (via common linker scripts) for intra-process memory isolation. These policies, as defined by the developer, consist of a set of rules codifying relationships between code and data, which are specifically the access controls (i.e. read, write, and execute permissions). ELFbac code within the Linux kernel then enforces these policies, via a finite-state machine where each state has a separate virtual memory context. The operating system kernel utilizes the information within the ELF file to link, load, and ultimately construct a runtime process.

Enforcement of an ELFbac policy happens in the kernel page fault handler. Memory accesses that happen outside the current state trigger page faults, which are then handled to verify if the memory access was valid at the current state or lead to an error state if the access was invalid.

We treat code and data units equally, and place fine-grained permissions over these units. We abstract the security-relevant phases of execution of a program into a finite-state machine (FSM). Permissions to code and data units are specified for each state, and the methods that trigger transitions are also specified. The permissions can be a combination of read, write, or execute. Sensitive data is placed in its own memory region, with read access only given when the data needs to be read. At all other states in the state machine, this permission is rescinded.

23.3.5 Implementation

Our team has built reference implementations for Intel x86 and ARM-based Linux systems. We believe that migrating ELFbac to other hardware platforms or similar operating systems should be made easier as portability is built into the current implementation. Wrappers for all the key kernel primitives were created to speed the porting process, so only a minimal set of files need to be changed.

Checking the policy by trapping the page fault on every memory access would cause an unacceptable drop in performance. To prevent this, ELFbac provides its own cache on top of the usual caching layers. To explain this, we need to first review how paging operates.

On the x86 platform, mapping is done through the page directory (PD) and the page table (PT), with each table containing 1024 4-byte entries. Each page directory entry (PDE) points to a page table, while each page table entry (PTE) points to a physical address; additionally, any page table lookup is cached by either the instruction or data translation lookaside buffer (TLB). Since the appropriate PDE and PTEs are created as needed by the running process and any system calls it makes, the page table entries serve as a proxy cache for the overarching policy in force in that memory space.

We add an additional layer of shadow memory where different code sections get different views of memory based on the policy FSM state. Adding additional contexts has the danger of "churning" the TLB, which would reduce performance; we counter this by filling the shadow contexts in a "lazy" fashion – every page starts empty and only when we have a valid access do we fill the shadow memory.

In practice, when the running process accesses an unmapped address, the Memory Management Unit (MMU) raises a page fault that ELFbac intercepts. ELFbac checks that the policy allows access and does not cause an FSM state change, and if true the shadow contexts are loaded so future access in the same state will not cause a fault. If the access causes a state transition, shadow contexts for the new state are loaded. The old memory context is unloaded so it can be validated against the policy on the next access.

Verification of the current state is done at each system call; the policy state can allow or disallow the call.

23.3.6 Using Mithril to Build Policies

ELFbac policies are written in a Ruby domain-specific language. Mithril embeds these policies in a separate `.elfbac` section in the binary, which is then enforced by the memory management system of the ELFbac enhanced kernel. We implement a policy for a parser using Mithril.

First, we build a state machine for the entire program. Figure 23.4 shows the state machine for our program. In the *start* state, all globals need to be given access since they are initialized. Following the initialization, we change state to the *parser* by invoking the *do_parse* method. The *libc* state handles access to the globals that are needed by the libraries. The variable being input by the user, the bytes received on the socket, the parse-tree built by the parser library, all need to be given access

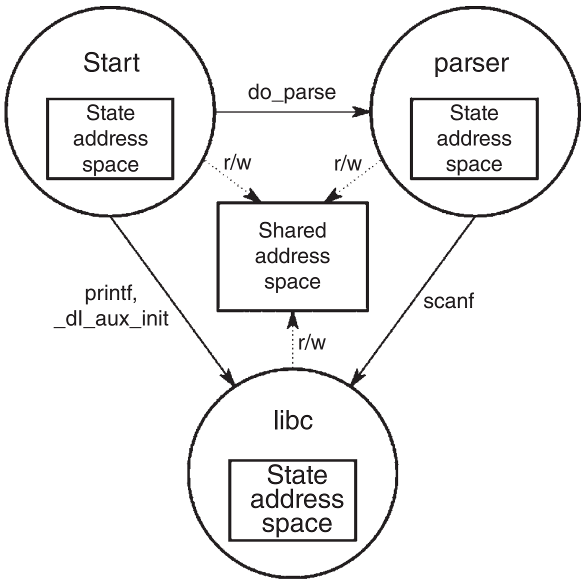


Figure 23.4 Building the state machine using Mithril.

in the *libc* state. This forces the developer to decide on the intentions for all the global variables and restrict their accesses between address spaces.

Second, we implement the state machine in the ruby-based domain specific language to reflect the origin state machine created. For each state, we specify the code sections that need to be given read, write, or executable access, and do the same for all the data sections. The function calls that trigger state transitions are also specified in the policy.

Third, we invoke Mithril to add the ELFbac section in the binary. We inspect the ELF binary to make sure the section was created. We trigger the code-paths that would violate our ELFbac permissions and cause *segfaults*. Exhaustive validation of all the possible state-transitions tells us that the policy is not over-permissive.

The syntax of Mithril involves keywords like: `tag`, `state`, `start`, `to`, `exec`, `readwrite` and `call`. Table 23.2 describes of each of these keywords.

23.3.7 How to Use ELFbac

While ELFbac requires some modifications to the OS kernel to enable policy enforcement, using ELFbac to preserve program intent is a straightforward process. Much of the work involved in creating an ELFbac policy revolves around modeling the security-relevant phases of execution of a program as a FSM and defining how the program transitions between these states at runtime. Here, we

Table 23.2 Syntax and usage for the keywords used in the Mithril Domain-specific language.

Syntax	Usage	Semantics
`start`	`start :main`	Set the start state for the state machine.
`exec`	`exec :code`	Specify that a code unit is only executable.
`readwrite`	`readwrite :data`	Specify the permissions for a data section.
`to`	`to :main`	Specify the state to transition to.
`call`	`call 'do_main'`	Specify a method that triggers a state transition.

`tag`	`tag :data do`
	`  section`
	`".datasec"`
	`end`
		Tag a section with a keyword. The tag can be used interchangeably with the section header.

`state`	`state :main do`
	`  readwrite :data`
	`  to :libc do`
	`   call 'scanf'`
	` end`
	`end`
		States specify code and data units and the transitions from those states.

give a tutorial on how to design and enforce ELFbac policies that align with developer intent to prevent vulnerabilities.

ELFbac is most easily applied during the software development process, as the developer can create the policy based on their existing knowledge of the domain and the programmer's mental model [16]. At this point, the designer and developer can work together to determine which pieces of code pose security risks and formulate a policy that isolates these pieces from the rest of the program. While it is possible to explicitly separate every symbol, function, or library from one another, not all of these interactions present a security risk. Proper identification of these risks helps reduce the burden on the policy creator and prevents unnecessary policy complexity.

Incorporating ELFbac into legacy programs is possible, but may pose more of a challenge if the original designer and developer are unavailable, as "familiarity with a codebase is essential to understanding potential areas of vulnerability" [16]. Therefore, extra care must be taken to understand the code and identify the pieces of code that need to be isolated [16].

Once the pieces of the program that require isolation are identified, we can define our FSM with the appropriate states our policy will need to achieve our isolation goals. Most programs already include this sort of structure implicitly: Most programs already include this structure implicitly: sections of code handling encryption, network communications, user input, etc. ELFbac supports state labeling at a number of different granularities, ranging from individual symbols to function boundaries to entire libraries [13], so states can be as small or as large as required by the user. The key is to ensure that any code and data that needs to be isolated is placed inside its own state within the FSM so that ELFbac can protect it from unauthorized accesses.

Once this policy is defined, the user can use an ELFbac-instrumented compiler to produce a policy-aware binary. The controls explained in Section 23.3.5 that ELFbac uses to isolate the requested program pieces are mostly hidden to the user, until an attempted policy violation causes the program to halt.

23.3.8 ELFbac in Action

We highlight two examples of how ELFbac works to mitigate intent-violating vulnerabilities:

- The OpenSSH roaming bug is an example of such a mismatch of intent. A malicious server can trick an SSH client into potentially sharing private keys and other data. We will demonstrate how ELFbac uses existing memory isolation techniques to enforce the principle of least privilege over separate code and data.
- Spectre variant 1 exhibits yet another mismatch of intents: The processor designer's intent for performance (via speculative execution) unexpectedly conflicts with the developer's intent to preserve secret data. We will also demonstrate how ELFbac naturally mitigates Spectre variant 1 via policy.

23.3.8.1 SSH Roaming Bug

In 2010, developers working on the OpenSSH client (an open-source program used to securely connect to other machines using the SSH protocol) added an experimental "roaming" feature to their software, aiming to allow clients to reconnect to a server in the case of an unexpected network loss [16]. To accomplish this goal, the OpenSSH client would store any messages that could not be sent to the server in a buffer, which the server could then request the contents of when the connection was reestablished. However, there was a major flaw: The server was not bound by the buffer's actual content and could request an arbitrary amount of data from it upon reconnection even if the client had not filled that space.

In 2016, CVE-2016-0777 described how a malicious SSH server could steal sensitive information from an OpenSSH client using this flaw in the roaming feature [17]. If the memory allocated for the roaming buffer had previously contained private SSH keys or other secrets and had not yet been erased or overwritten by the client, it could be stolen by the server via a malicious buffer request [16]. While the risk of this vulnerability was limited to connections to compromised SSH servers, its existence demonstrated a clear violation of programmer intent: The feature was not intended for production release; the developers did not intend to give the server control over what data could be sent, and the developers certainly did not intend for sensitive information to be exposed via the roaming buffer.

To mitigate this flaw, we need to isolate the code and data involved in cryptographic operations from that which handles network operations, thus ensuring that secret keys cannot be leaked via the roaming buffer [16]. Since this is an existing codebase, we first needed to perform a thorough review to identify the pieces of the program that are involved in these tasks and thus need to be isolated.

Our resulting FSM consists of three states: A *crypto* state for tasks involving private and secret SSH keys, a *network* state that deals with network communications, and a *default* state that encompasses all other operations. (For the sake of simplicity, we assume that there are no other code/data interactions that pose a security risk; however, our existing states could be further split into smaller ones if the need arose.) Each state maintains its own copy of the process address space, but also has access to a shared address space in case data needs to be passed between states. Figure 23.5 depicts our full state machine to apply ELFbac to the OpenSSH Roaming bug.

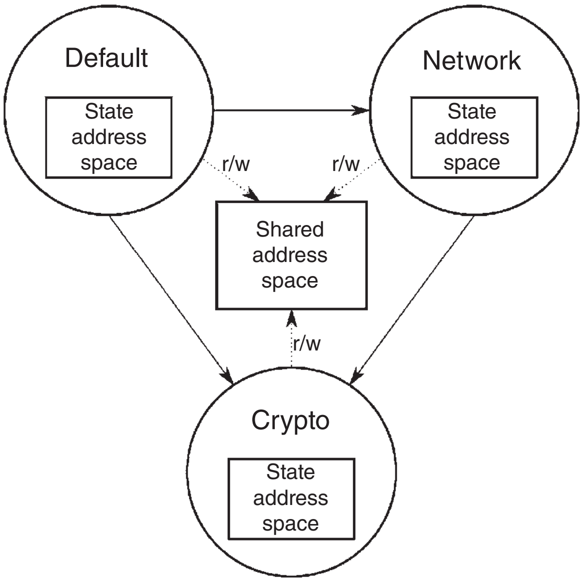


Figure 23.5 The finite state machine represented by our OpenSSH ELFbac policy. The memory allocated for private keys and the roaming buffer will come from different heaps within different address spaces, keeping the keys from being leaked to a malicious server.

The key insight here is that each state will maintain a private heap within its address space, which it uses for allocating memory for its own data items [16]. This separation of heaps means that even if a private key is not removed from memory after use, it will never overlap with the program's roaming buffer because the two items do not share the same memory space. The server can request as much of the buffer as they want, but any data used for cryptographic operations remains out of reach, thus mitigating the roaming bug.

23.3.8.2 Spectre Variant 1

In 2018, researchers revealed two major security flaws in the architecture of nearly every processor chip used in the last couple decades. One of these vulnerabilities was dubbed Spectre [2]. To achieve greater efficiency, modern processors are highly parallel in their operation. This parallelism inside instruction execution is often termed *pipelining*, and as an optimization technique, it allows the various execution and memory units of a CPU to operate simultaneously. A challenge for pipelining is conditional branching within a program. What code and data should be loaded into the pipeline when a branch is encountered? If the wrong decision is made, the pipeline must be flushed (or cleared away) so that the correct code can properly execute. Branch prediction is an additional optimization technique useful in limiting the amount of pipeline flushes that occur during execution. In essence, through various heuristics, a processor guesses which direction the conditional branch will take. Speculative execution is the pipelining and execution of instructions, after a conditional branch, based on a processor's guess about how the conditional will evaluate.

Spectre attacks take advantage of this branch prediction and the latent architectural effects of speculative execution. Kocher et al. show that "... speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks" [2]. Here we find the violation of intents:

- Processor designers did not intend for there to be any noticeable, latent architectural effects.
- Operating system developers assumed their understanding of the architectural pipeline flushing mechanisms.
- Software developers intended for their program's secrets to be unaffected by spurious conditional branches.

To see this in action, Kocher et al. provide a proof-of-concept in C [2] that demonstrates an adversarial training of the branch predictor and its subsequent exploitation to reveal "secret" data that is never actually read during normal program operation. A snippet of the lines of interest follows in Listing 23.1.

The code begins with the creation of several regions of memory, e.g. `array1` and `array2`, including a `secret`. Additionally, there is a `victim_function` with a key conditional branch. To train the branch predictor, the PoC calls this function with valid values of input parameter `x`. These values touch areas of memory in `array1` and `array2`. After training, the `victim_function` is called with a malicious `x`. This malicious `x` should cause the conditional branch to evaluate false. Because of the prior training, the code is executed speculatively, which loads the page table entry that contains the secret string, thus caching the secret. After the secret is cached, the branch conditional is evaluated, and the CPU unrolls the speculatively executed instructions. However, this leaves behind the secret in the cache. Via some side-channel timing methods, this secret can be extracted.

A key observation is that the secret is never touched during normal execution of the program. The time when it is loaded is essentially by proxy, grouped along with other data within a page table entry. However, the latent effects of the architecture, i.e. the caching mechanisms, allow this data to be extracted. We draw the analogy of breadcrumbs being left behind during program execution.

So, what can ELFbac do to bridge this intent gap? ELFbac's primary means of policy separation is using page table differentiation. So, naively, the solution to this particular PoC, from the ELFbac perspective, is to place the secret on a separate page table.

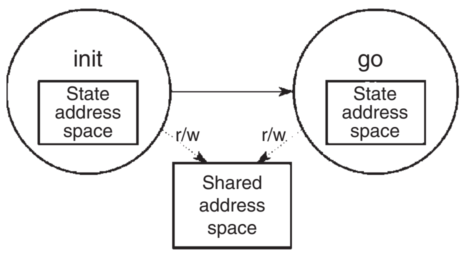
Listing 23.1 Spectre PoC from Kocher et al.

```
/****************************
Victim Code
****************************/
uint8_t array1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */
void victim_function(size_t x) {
    if (x < array1_size) {
      temp &= array2[array1[x] * 512];
      }
}
```

(a)



(b)

```
Elf::Policy.inject_symbols (file)
    x = Elf::Policy. build do
      state 'init_state' do
        readwrite :default
          exec :default
        readwrite :secret
         to 'go state' do
            call 'go'
      end
   end
   state 'go_state' do
      readwrite :default
      exec : default
      exec :go
    end
    Start 'init_state'
  end
  x.inject(file)
```

Figure 23.6 (a) State machine. (b) Mithril policy that mitigates Spectre variant 1.

As seen in Figure 23.6, two states are modeled. The `init` state represents the entire program with access to the `secret` data. Any access to the `secret` variable requires a state-transition to the `init` state. Outside of `init`, the `secret` is not accessible. As seen in part (b), only the `init_state` has `readwrite : secret`.

The key insight here is that page-table permissions are respected by the architecture. When the branch predictor is being trained, the `secret` variable is never "gobbled" by the paging mechanisms and thus cached. **When the speculative instructions are pipelined and executed, a page fault is raised when access to `secret` is requested in the wrong state.** This causes the pipeline to be stalled, voided, and eventually cleared before the caching mechanisms are invoked.

23.3.9 Summary

As IoT devices scale into the billions, it is absolutely imperative that they do not fall victim to intent violations stemming from vulnerabilities such as those described above. ELFbac is an effective, powerful policy solution that provides a critical layer of security against attackers looking to subvert designer and developer intent. It can be applied to both new and existing programs, and, for long-lived IoT devices where traditional security solutions (such as regular patching) are not feasible, ELFbac can still help protect against zero-day vulnerabilities and keep them from being exploited.

23.4 Building a Secure Implementation of AMQP

As an example of using LangSec and ELFbac together for the IoT, we consider hardening of AMQP protocol implementations.

Demonstrating that an implementation of a network protocol is secure from crafted-packet vulnerabilities includes ensuring that the parser rejects every message that does not conform to the grammar. Another implementation property to ensure: If a vulnerability that can be exploited exists in the parser, then it is isolated and placed in a separate memory region, so it does not affect the rest of the memory. (This example applies LangSec *and* ELFbac to the parser. Another family of use cases would apply LangSec to the parser and then apply ELFbac to the main program to contain damage from any attacks that get through this input validation component.)

The Advanced Message Queuing Protocol (AMQP) is a networking protocol that clients can use to communicate with each other through the server. AMQP requires at least one server (also known as a broker), and more than one client. The server consists of various components: an exchange, a message queue, and a binding. The exchange receives messages and decides which message queue must receive the message. Message queues store the messages sent if the messages haven't been consumed yet. Bindings specify which received message is put in which message queue. Traditionally, the clients *subscribe* to channels on the broker. Producers publish messages that are received by consumers.

The AMQP protocol is not very suitable for devices with very limited memory because of the size of the optional fields, but it is very suited for industrial IoT use cases such as SCADA systems with reliable network connectivity and bandwidth. Having an asynchronous queuing system for data means that the data will always be eventually processed, despite traffic spikes or network connectivity issues. AMQP supports better security protocols than its predecessor MQTT and also supports federation of various AMQP servers. Support for such delegation and a resilient security architecture make AMQP one of the most popular IoT protocols. A search for the AMQP port 5672 on Shodan shows that there are over 900 000 AMQP brokers operating on the Internet worldwide [18]. More than half of these AMQP brokers are operating in the United States alone. We also studied the number of reported vulnerabilities in implementations of the AMQP protocol. Table 23.3 shows that a significant portion of the number of reported vulnerabilities comprise of parser bugs which could be avoided using LangSec and isolated with the use of ELFbac.

In this section, we discuss the application of two tools: First, we use the Hammer parser-construction toolkit to demonstrate its efficacy on the AMQP protocol. Second, we implement and inject an ELFbac policy using Mithril.

Table 23.3 Number of vulnerabilities reported by the common vulnerabilities and exposures database maintained by Mitre for the AMQP protocol.

Year	Number of input-handling vulnerabilities	Total number of vulnerabilities
2015	2	3
2016	2	3
2017	4	5
2018	3	5

23.4.1 A Deeper Understanding of the AMQP Protocol

The various messages spoken by the client and the server in the AMQP protocol are demonstrated in Figure 23.7. The connection is initiated by the client, whereas the server is waiting to receive a communication.

The messages received by the broker are: Connection Start, Connection Tune, Connection Open, Channel Open, Queue Declare, Basic Publish, Basic Get, Channel Close, and Connection Close. The messages sent by the broker are: Connection Start OK, Connection Tune OK, Connection Open OK, Channel Open OK, Queue Declare OK, Basic Publish OK, Basic Get OK, Channel Close OK, and Connection Close OK. The receiving parser on the client needs to be able to recognize all the messages that are sent by the broker, and the parser on the broker needs to be able to recognize all the messages sent by the client. The state machines of the client and the servers are shown in Figures 23.8 and 23.9.

As a part of the LangSec approach to implementing parsers we analyze individual packet formats for the various messages supported by the protocol and extract the syntactic features used by the protocol. As seen in Figure 23.10, the packet format uses a length field that must be parsed to correctly parse the rest of the packet. The length of the rest of the packet must be equal to the length field in the packet.

The *payload* of an AMQP packet can be of various types: method frame, content-header frame, content-body frame, and heartbeat frame. The method frames are used the most, and the syntax structure of the method frame can be found in Figure 23.10.

As we mentioned earlier, LangSec argues protocols should be either regular or deterministic context-free. We consider all the structural requirements for the payload fields of the various packet formats in the AMQP protocol. We do not impose syntactical restrictions only on the headers but the entire packet as a part of the LangSec methodology. The AMQP specification includes a generic grammar for the AMQP packet format. Although the header usually remains similar across

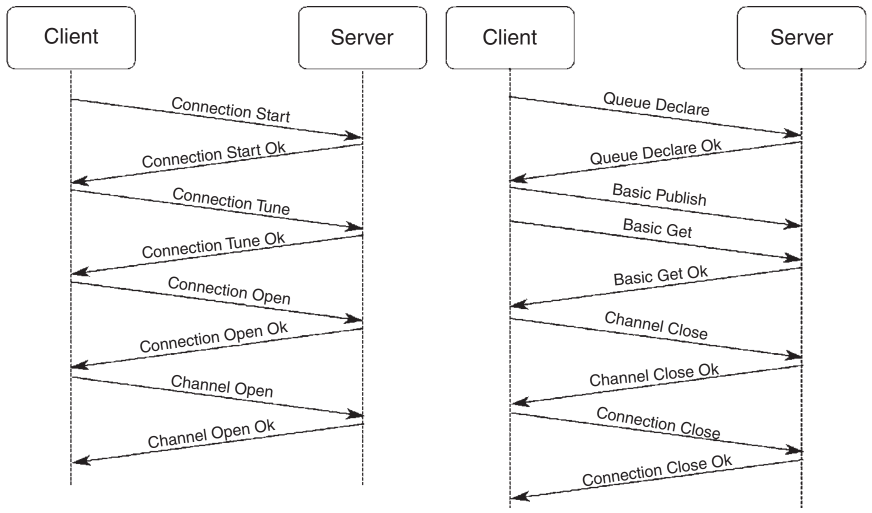


Figure 23.7 Flow of messages in the AMQP protocol.

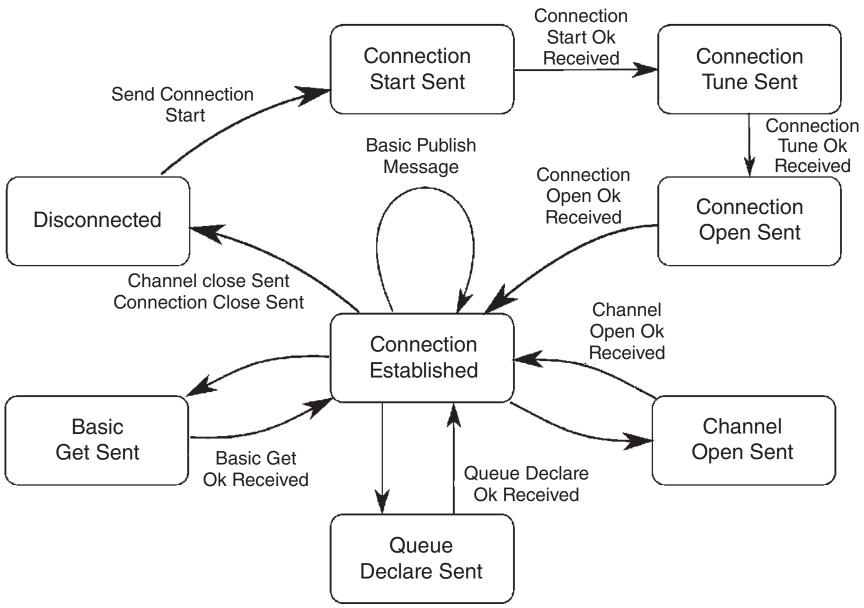


Figure 23.8 AMQP Client state machine. The client initiates the connection with the broker by sending a Connection Start message.

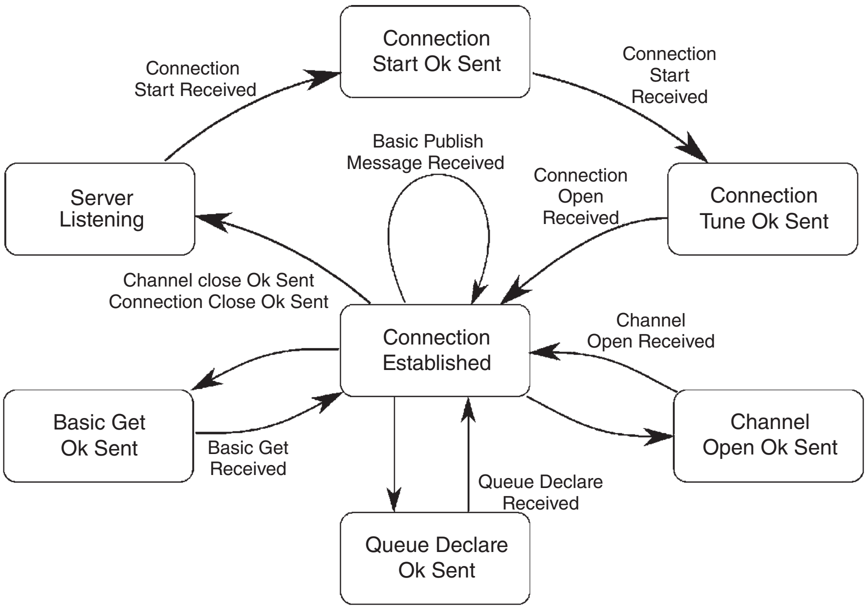


Figure 23.9 AMQP Broker State Machine diagram showing the various states and transitions. The server responds to connection initiation requests from clients.

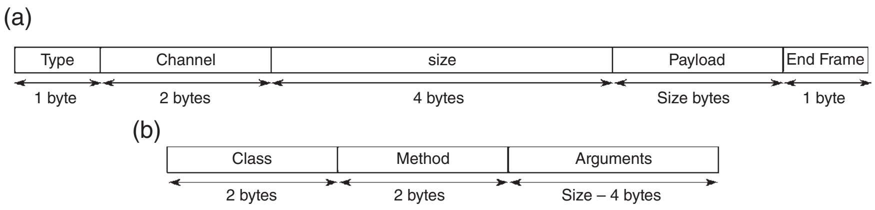


Figure 23.10 (a) Packet format of a generic AMQP packet. The number of bytes in the payload depends on the "size" bytes. (b) Format of the method frame used in AMQP payloads.

the different packet formats of AMQP, having separate grammars for the various packet formats would improve readability and make the parsers more fine-grained.

23.4.2 Hardening AMQP Parsers with Hammer

In the previous section, we detailed the AMQP protocol and its various message formats. Although the sequence of messages is important for any protocol filter, the contents of the messages aren't completely related, i.e. a parser built for the

Connection Tune message format does not rely on fields from its previous message, which was *Connection Start*. In case of protocols with reliance on data from previous message, we advocate performing stateful parsing, after checking for *well-formedness*. A well-formedness check would only ensure the structural correctness of the header, without checking for the equivalence of the reliant fields. This well-formedness check is followed by stateful parsing, where the fields are validated with respect to the data previously received.

Once we extract the syntax for the various messages in a particular protocol, we begin implementing our parsers for the protocol. The AMQP header begins with the `type` field, which is 8 bits long. There are a limited number of values that can comprise of the `type` message. For example, the field can only comprise of the values 1 through 4 corresponding to method frames, content header frames, content body frames, and heartbeats. We witness that the method frames are used most frequently in our data traces of the AMQP protocol. Apart from examining the sizes of each of the fields, we are also imposing constraints on the data itself – ranges of integers, ranges of characters, choices of strings, etc.

The syntaxes for each of the message formats are converted to specific parsers for each of the states in the state machine. We use `scapy` to extract the application layer of the packets and pass it on to the AMQP parser. The AMQP parser is implemented as a Docker container that receives raw messages and validates the syntax of the messages. The design of our parser implementation is in Figures 23.11 and 23.12.

23.4.3 Implementing a Policy for AQMP in Mithril

We implement a policy for the AMQP parser in the Mithril domain-specific language, already described in Table 23.2. First, the state machine is extracted for an implementation of the AMQP protocol. The states for the program as a whole are: start, parser, and libc. The globals are initialized in the start state, the message is received in the buffer. The buffer is then passed on to the parser. The parser

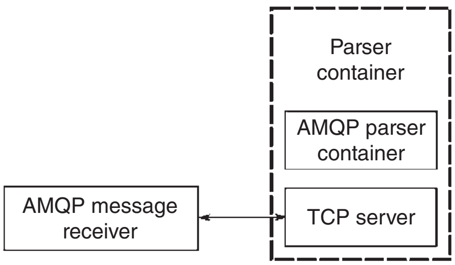
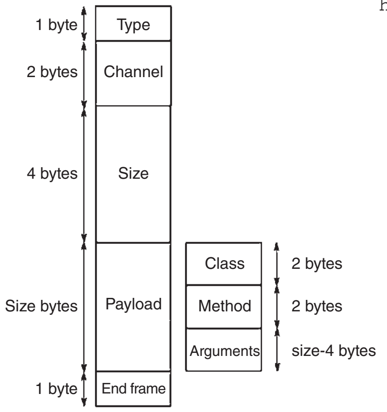


Figure 23.11 Design of our AMQP parser implementation.



```
h_sequence(
   h_uint8(),   # Type
   h_uint16(),  # channel

    h_and(h_length_value(
        h_uint32(),
        h_ch_range('\x00', '\xff'))),
    h_uint32(),     # Size
    h_sequence(
        h_uint16(), # class
        h_uint16(), # Method
        h_optional(arguments)), # Arguments
    h_ch('\xce'),  # End Frame
    h_end_p(), NULL);
```

Figure 23.12 Conversion of a syntax structure for a packet format into Hammer parser-combinator code.

performs the parsing operation and hence has to have access to the necessary globals. In the libc state, the program performs operations such as `scanf` and `printf`.

Second, this state machine is then converted into a Mithril policy. Each state is specified with a fine-grained policy specifying all the global variables that need to be accessible and the transitions that would lead to that state. Our policy for AMQP starts with the `start` state and then transitions to the parser when the `do_parse` method is called. All globals need to be given `readwrite` access to the `start` state since the globals need to be initialized. Since Mithril makes use of dynamic loading, we transition to `libc` whenever `_dl_runtime_resolve` is called. Since in our implementation we do not make use of any other libraries, this method would only be executed when functions from `libc` are called.

The state machine and the corresponding Mithril policy code can be found in Figure 23.4.

23.5 Evaluation Techniques

Thus far in this chapter, we have studied existing research to enforce the programmer's intent with techniques such as LangSec and ELFbac. In this section, we will survey some of the techniques widely used to test software built on the principles of LangSec and ELFbac.

23.5.1 Coverage-Guided Fuzzing with AFL

American Fuzzy Lop (AFL) is a state-of-the-art coverage-guided fuzzer that is protocol agnostic. It takes a set of samples and runs genetic algorithms on it to generate more samples to run through a program [19]. Fuzzers run over their target many times with different inputs. AFL does not require that we extract the specific target methods into separate files and then compile those files as fuzz targets.

Other techniques, such as control-flow analysis and data-flow analysis, demonstrate interesting ways to understand how a binary operates and what operations the binary performs. However, they do not scale well for large applications. AFL can be given a single binary, and it can perform brute-force fuzzing that covers all edges of a program's control flow.

```
afl-fuzz -i testcase.dir -o findings.dir /path/to/program @@
```

Binaries are fuzzed with the script above. AFL provides a text-based GUI to display the progress, number of paths discovered, number of crashes and hangs, and the run time. The inputs that lead to crashes and hangs are all logged in separate folders.

23.5.2 Static Analysis Using Infer

Infer is a static-analysis tool that can be run on C code to detect certain categories of errors commonly found. Namely, Infer can catch null de-references, memory leaks, premature nil termination arguments, and resource leaks [20]. Static analysis provides another layer of assurance that the program doesn't have any vulnerabilities.

Static analysis requires access to the source code. These tools analyze all code paths of a program and check for paths that lead to crashes, memory violations, memory corruption, memory leaks, etc. They can also be programmed to detect style violations. In our implementations, we use [20] and make sure there are no memory violations.

Memory violations usually indicate poor handling of input, which could lead to vulnerabilities. Capturing programmer's intent with LangSec and ELFbac can alleviate these memory violations and isolate them.

23.6 Conclusions

In this chapter, we introduced the notion of intent as a secure design primitive and we discussed the importance of preserving designer and developer intent for system security. We presented two security paradigms to safeguard against adversaries who wish to subvert system security by violating designer and developer intent: *LangSec*, which constrains program inputs to safe options as defined by the protocol specification and the Chomsky hierarchy, and *ELFbac*, which isolates unrelated code and data and enforces boundaries between program pieces as defined by designer/developer policies. Finally, we showed how a simple program such as an AMQP parser could be hardened using tools delivered by these paradigms, and we introduced various ways to test and validate such hardened programs.

While LangSec and ELFbac provide some mechanisms to preserve designer and developer intent, work remains. For instance, we aim to extend LangSec's functionality beyond just validating packets against a protocol specification; we intend to start examining the data within the packet to ensure that it makes sense within the program's context. For example, can we examine the metadata around a piece of data (where it came from, what values we have seen in the past, etc.) to make a decision about how trustworthy it is? With ELFbac, we are looking for ways to formally prove specific security properties of ELFbac-enhanced programs, similar to projects such as Low* [21].

We also intend for our tools to extend beyond merely enforcing intent. We aim to help the designer and the developer think critically about their intentions and what they should or should not do to maintain system safety. Currently, LangSec and ELFbac blindly follow the instructions of the designer and developer and blindly ensure that whatever intent they express is not violated. However, if the original intent of the system is misguided (for example, if the desired protocol is too powerful to be verified as safe), our tools should alert the designer/developer to this issue and help them produce a better alternative that balances safety with functionality. At the very least, we want LangSec and ELFbac to make designers and developers explicitly consider their intent and ask themselves if they are asking too much of their system.

Finally, we must also consider preservation and refinement of user intent, which neither LangSec nor ELFbac currently address. As we noted before, users are ultimately the ones who determine how a system operates, and thus their actions can supersede the intent of the designer and developer. As a community, we must pay special attention to the *usability* of systems to ensure that the user uses the system the way the designer and developer intended, instead of violating designer and developer intent to achieve their primary task.

In sum, securing the IoT requires realizing the intent of each stakeholder. LangSec and ELFbac make great strides toward addressing this grand challenge, but more work must be done.

23.7 Further Reading

For an in-depth discussion of the theoretical aspects of LangSec, we recommend the reader consult Sassaman et al. [22]. For Linux kernel implementation details of ELFbac, we recommend consulting Bangert et al. [13].

The yearly LangSec Workshop at the IEEE Symposium on Security and Privacy provides a venue to discuss LangSec research ideas. For case studies of LangSec and ELFbac, the reader may consult Anantharaman et al. [23], Bratus et al. [24], and Jenkins et al. [16].

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-16-C-0179, ONR under grant N00014-18-1-2022 and Department of Energy under Award Number DE-OE0000780.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force, DARPA, ONR, United States Government, or any agency thereof.

References

1 M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis et al., “Understanding the Mirai Botnet,” *in Proceedings of the 26th USENIX Security Symposium,* USENIX Association, 2017, pp. 1093–1110.

2 P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre Attacks: Exploiting Speculative Execution,” In Proceedings of the *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, IEEE Computer Society, 2019, pp. 1–19.

3 Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. Alex Halderman. “The Matter of Heartbleed,” In *Proceedings of the Conference on Internet Measurement Conference (IMC ’14)*, Association for Computing Machinery, New York, NY, USA, 2014, pp. 475–488.

4 T. Fox-Brewster, “What Is the Shellshock Bug? Is It Worse than Heartbleed,” *The Guardian*. https://www.theguardian.com/technology/2014/sep/25/shellshock-bug-heartbleed.

5 P. Ducklin, “Anatomy of a Security Hole – Google’s “Android Master Key” Debacle Explained,” *Naked Security*, Sophos. https://nakedsecurity.sophos.com/2013/07/10/anatomy-of-a-security-hole-googles-android-master-key-debacle-explained/.

6 F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, “The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them,” in *Proceedings of the 1st IEEE Cybersecurity Development (SecDev)*, IEEE Computer Society, 2016, pp. 45–52.

7 S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina, “Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation,” *USENIX; login:,* 2011, vol. 36, no. 6, pp.13–21.

8 M. Sipser, *Introduction to the Theory of Computation*, Thomson Course Technology, Boston, 2006.

9 G. Sénizergues, “The Equivalence Problem for Deterministic Pushdown Automata is Decidable,” in *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, Springer-Verlag, Berlin, 1997, pp. 671–681.

10 J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems,” In *Proceedings of the IEEE*, IEEE Computer Society, 1975, vol. 63, no. 9, pp. 1278–1308.

11 J. Postel, “DoD Standard Transmission Control Protocol,” *ACM SIGCOMM Computer Communication Review*, ACM, 1980, vol. 10, no. 4, pp. 52–132.

12 L. Sassaman, M. L. Patterson, and S. Bratus, “A Patch for Postel’s Robustness Principle,” *IEEE Security & Privacy*, IEEE Computer Society, 2012, vol. 10, no. 2, pp. 87–91.

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, Department of Computer Science, Technical Report 727, June 2013.

14 P. A. Loscocco and S. D. Smalley, “Meeting Critical Security Objectives with Security-Enhanced Linux,” in *Proceedings of the Ottawa Linux Symposium*, 2001, pp. 115–134.

15 E. Witchel, J. Cates, and K. Asanović, “Mondrian Memory Protection,” in *10th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS ’02*, ACM, 2002, pp. 304–316.

16 I. R. Jenkins, S. Bratus, S. Smith, and M. Koo, “Reinventing the Privilege Drop: How Principled Preservation of Programmer Intent Would Prevent Security Bugs,” in *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security, ser. HoTSoS ’18*, ACM, 2018, pp. 3:1–3:9.

17 CVE-2016-0777. MITRE, January 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0777.

18 Shodan. “Shodan: The Search Engine for the Internet of Things.” https://www.shodan.io/.

19 M. Zalewski, “American Fuzzy Lop,” 2015.

20 C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving Fast with Software Verification,” in *Proceedings of the 7th International Symposium NASA Formal Methods Symposium*. Springer International Publishing, 2015, pp. 3–11.

21 J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţçu, K. Bhargavan, C. Fournet, and N. Swamy, “Verified Low-Level Programming Embedded in F∗,” in *Proceedings of the ACM on Programming Languages*, ACM, 2017, vol. 1, no. 17, pp. 1–29.

22 L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, and A. Shubina, "Security Applications of Formal Language Theory," *IEEE Systems Journal*, IEEE Computer Society, 2013, vol. 7, no. 3, pp. 489–500.

23 P. Anantharaman, M. Locasto, G. F. Ciocarlie, and U. Lindqvist, "Building Hardened Internet-of-Things Clients with Language-Theoretic Security," in *Proceedings of IEEE Security and Privacy Workshops (SPW)*, IEEE Computer Society, 2017, pp. 120–126.

24 S. Bratus, A. J. Crain, S. M. Hallberg, D. P. Hirsch, M. L. Patterson, M. Koo, and S. W. Smith, "Implementing a Vertically Hardened DNP3 Control Stack for Power Applications," in *Proceedings of the 2nd Annual Industrial Control System Security Workshop*, ACM, 2016, pp. 45–53.