Author manuscript, published in "IEEE/IFIP Network Operations and Management Symposium - NOMS 2010 (2010) 216-223"
DOI : 10.1109/NOMS.2010.5488464

inria-00544976, version 1 - 9 Dec 2010

VeTo: An Exploit Prevention Language from Known Vulnerabilities in SIP Services

Abdelkader Lahmadi * and Olivier Festor†
*LORIA - Nancy University - INPL, Villers-Lès-Nancy, France
†INRIA Nancy - Grand Est Research Center, Villers-Lès-Nancy, France
Email: {Abdelkader.Lahmadi,Olivier.Festor}@loria.fr

***Abstract*—We present VeTo a language to specify protection rules for VoIP systems, supported by the *SecSip* prevention framework. VeTo offers a unique way to specify both vulnerabilities and countermeasures to protect SIP services against known vulnerabilities. We illustrate the applicability of the language through the specification of several known attacks and assess its efficiency through a target testbed.**

***Index Terms*—SIP, Vulnerability, Language, Prevention**

I. Introduction

The Session Initiation Protocol (SIP) [16] is designed to establish, modify, and terminate a session of application services. SIP security has recently gained an increasing interest and is becoming a hot topic both in academic and industrial research. This is due to the large number of discovered vulnerabilities related to the SIP protocol. The sources of these vulnerabilities are mainly the weakness of some of its implementations and sometimes even its specification semantics [14]. Implementation dependent vulnerabilities are usually discovered using a fuzzing process [1] where the main purpose is to discover the impact of either malformed messages or the impact of a specific sequence of messages on a particular SIP implementation. Specification based vulnerabilities can be found either through formal analysis of SIP related specifications and can be validated using attacking tools.

A primary way to counter SIP implementation vulnerabilities is through patching. However, patching time is often important [15] and until that happens the SIP network is kept on leash to attackers. To protect SIP services from such scourge, a first line of defense systems have to be deployed. A defense system can start with network level firewalls, where packets are filtered without a deep understanding of the SIP protocol semantics. Another way, is to use detection engines like *Snort* [3] with a set of known attack signatures.

Recently, several SIP aware application layer firewalls have been developed to support attack mitigation on SIP networks. The proposed solutions to protect SIP networks from known vulnerabilities lack sufficient flexibility and extensibility. A prevention system needs to be fed with vulnerabilities specification and their counter measures using a domain-specific language. The prevention system runtime screens the SIP traffic and identifies vulnerabilities according to the authored specifications. Most of existing protection systems use low-level languages such as C, or plugins to statically load vulnerabilities specification. The authoring process must be repeated for each new discovered vulnerability, despite the fact that many SIP vulnerabilities share common properties over SIP messages, dialogs or fields.

In this paper, we present a domain-specific language [13], called VeTo, that is used to specify preventions schemes for known vulnerabilities of SIP based services such as Voice-over-IP. VeTo is an event-driven rule-based language to describe SIP vulnerabilities and their respective countermeasures. It is stateful since it includes features to record SIP protocol states over messages and dialogs. Each specified prevention scheme is executed by a runtime engine deployed as "a bump in the wire" device in a SIP network. In previous work [11], we have presented a tool called SecSIP which intercepts SIP traffic and is capable of executing VeTo rules.

The remainder of the paper is organized as follows: Section II presents existing attack languages in the context of network intrusion detection with a focus on the STATL [6] and the SHIELD [22] languages. Section III, describes the main known vulnerabilities in the SIP protocol and its implementations. In section IV, we detail the VeTo language features and semantics. We illustrate its use on two examples: vulnerabilities behind malformed messages and flooding attacks. Section V provides a qualitative analysis of VeTo, Snort and STATL to express different prevention schemes. We then evaluate the SecSIP runtime using basic scenarios to assess its call handling capacity with activated preventions. Finally, in section VI we draw some conclusions and future works.

II. Related Work

To our knowledge, no dedicated description language has been published so far to encode SIP attacks or vulnerabilities and the respective countermeasures. However, main research activities were focused on the development of runtime systems and mechanisms where vulnerability occurrences and attack signatures are encode internally. In the general area of network intrusion detection mechanisms, several attack and detection languages have been proposed [20].

The VeTo language design benefits from the exhaustive literature on the development of attack and vulnerabilities description languages. We identify two classes of such languages: stateful and stateless languages. Stateless languages describe attack events independently, while stateful languages consider the relationships between events and are able to model event "histories" that represent attacks. Previous existing languages

inria-00544976, version 1 - 9 Dec 2010

do not provide features to address SIP protocol semantics for vulnerabilities description. Instead, they only use generic signature models, e.g., applying regular expression or state machines to a known protocol data. The generated signatures are usually independent and inefficient when the number of specified vulnerabilities becomes important.

A. *Stateful languages*

Languages based on finite state machines such as STATL [6] and SHIELD [22] have been designed to express network protocol attacks and vulnerabilities. The SHIELD is a vulnerability driven language to express application layer binary protocol vulnerabilities at end-hosts. In their approach, the authors only specify the part of the protocol state machine that exhibits a vulnerable behaviour. VeTo shares with SHIELD this feature since our aim is to only specify the part of the SIP protocol involved in the vulnerability. Due to the limited data available in the SHIELD paper, its applicability to SIP could no assessed by us.

GAPAL [2] is a generic application level protocol analyser that relies on a stateful language to describe protocol syntax and semantics for their parsing and analysis. Both the GAPAL and SHIELD languages have the same syntax, the former being an extension to the later to support the analysis of text-based protocols.

In [6], the authors present a state/transition based language, called STATL, dedicated to attack recognition. The language allows the description of domain-independent attacks. It relies on the state transition analysis technique [9] which is an abstraction of attack actions to a higher representation. In STATL, an attack is modeled as a sequence of steps that bring the system from an initial safe state to a compromised state. The attack is represented as a composition of *states* and *transitions*. They also use variables to record the part of the system state needed to define an attack signature. Transitions have an associated action that is the event that may cause the transition to a new state. The language has been used on different domains such as network attacks and Unix system attacks. Unlike STATL, the VeTo language is not a pure state/transition based language, but it uses an event-driven approach. An event-driven approach is more suitable to overcome the STATL limitations regarding time, intervals and relationships among events. Only, representing events as transitions through a single chain of states precludes the recognition of a vulnerable behaviour without any time order. In addition, a pure state/transition approach is ineffective [18] and may introduce a state explosion problem when describing a large number of vulnerabilities.

B. *Stateless languages*

Snort [3], the defacto-standard open source intrusion detection tool relies on a signature-based language to detect intrusion. Its language is signature based and stateless since it does not provide means to record protocol states. The stateful feature of Snort is provided by the preprocessors, where the stateful inspection is hard coded. For example, the *stream4* preprocessor is used by Snort to maintain an internal state table for each TCP session. The VeTo language provides an expressive way to construct stateful variables which keep track of SIP protocol histories. In addition, the language provided by snort is signature-based and relies on exploit description rather than the vulnerability description approach used by the VeTo language.

Extending these existing languages to support SIP semantics do not resolve these issues since their underlying matching models are hardcoded. Therefore, we propose a novel language dedicated to the prevention from existing vulnerabilities of the SIP protocol.

III. Motivation and Background

SIP [16] is a protocol designed to negotiate, establish and terminate a session between two or more peers. It is used as a signaling protocol to provide many services such as voice. A SIP service infrastructure relies on several entities, including a *User Agent* that generates or terminates SIP requests, registrars, where users register themselves and announce their availability in the SIP network and proxies that forward requests in the appropriate SIP networks. Despite, the well-known attacks that the SIP architecture inherits through the utilisation of Internet technologies, there are specialised attacks on the SIP protocol itself or on its implementations. The attacks exploit existing known errors in SIP implementations or weakness in its design.

A. *Sources of SIP protocol vulnerabilities*

SIP messages are text-based and use the *UTF-8* charset. This feature leads to an easy modification by man in the middle attackers. Cryptographic mechanisms to protect SIP messages are not widely deployed because of intermediate nodes that may modify SIP messages for their routing. The SIP protocol uses the *INVITE* message to setup and modify dialogs over their lifetime. According to the RFC 3261 [16], the message used to modify dialog properties is called *re-INVITE*, but it has the same method value as INVITE. This may results in a theft of service by exploiting a spoofed or simply relaying a crafted *re-INVITE* message to establish calls.

The SIP protocol provides a forking proxy feature to allow the setup of multiple dialogs from a single request message. This leads to branch an incoming request to multiple outgoing requests, each targeted to a different User Agent Server (UAS). Recently, RFC 5393 [19] proves that this feature may cause a massive flooding attack with valid SIP requests between proxy-to-proxy messages. A version of this attack demonstrates that less than ten messages can lead to the generation of potentially 2^{71} messages.

SIP routing relies on the proxy elements that take downstream routing decisions based on the routing headers and upstream routing decisions based on *Vias* fields. The SIP routing mechanism leverages the following vulnerabilities [17]:

- Any element can insert/delete/alter routing headers.
- Proxies route statelessly without call-route state or global route knowledge.

inria-00544976, version 1 - 9 Dec 2010

- There is no authoritative, trusted element for end-to-end routing policy enforcement.
- SIP privacy directives allow for route information stripping. As such, there is no way to differentiate legal or malicious deletion or replacement of routing headers.

These vulnerabilities may generate different types of attacks. These attacks are broken down into network topology privacy breach, toll fraud and DoS based attacks.

As described in [16], SIP mandates the use of HTTP digest-based authentication as a security mechanism to protect SIP messages. It provides anti-replay protection and one-way authentication to SIP messages. However, the SIP authentication mechanism has the following weaknesses [24]:

- It is not an end-to-end security model since SIP servers and proxies need to investigate and change certain fields.
- It only applies to a few SIP messages (INVITE, BYE, REGISTER), and leaves other important SIP messages (TRYING, OK, ACK, BUSY) unprotected.
- It only protects a few SIP fields (Request-URI, realm), and leaves other important fields unprotected (e.g, From, To).
- It only applies to SIP messages between the User Agent Client (UAC) and SIP servers, while leaving all SIP messages from the SIP servers to UAC unprotected.

As described in [24], these vulnerabilities can be exploited for thieve of service attacks.

B. Complexity of fixing vulnerabilities

In this section, we draw attention to a few key aspects of fixing known SIP vulnerabilities, particularly their complexity to patch and to manage.

a) Rising number of vulnerabilities and reluctant administrators: The major problem facing the fixing of vulnerabilities is their rising number and the ability of users to cope with the number of patches issued for these vulnerabilities [10]. As this number rises, the time to patch becomes important up to the order of months [15] . In addition, patches can be difficult to apply and might even have unexpected side effects as a result of compatibility issues.

b) Diversity and dynamics of vulnerabilities: Discovered vulnerabilities in the SIP world may involve different dialogs and different headers over different messages. Our analysis of existing vulnerabilities shows that they may share the same SIP protocol properties but for different purposes. For example, many malformed SIP message vulnerabilities have the same type of messages but rely on different malformed headers like via, call-ID, or content length. A vulnerability has a lifetime dynamics where it may be discovered or disappear after a patching operation or a software upgrade. Therefore, the prevention mechanism requires a configuration changes to reflect the evolving nature and the emergence of new vulnerabilities.

c) Multiple vulnerabilities per device: A device involved in a SIP service has usually its own vulnerabilities due to errors or weakness in its implementation. The prevention from the exploit of these vulnerabilities has to take into account that one or many vulnerabilities are associated to a particular device. The prevention mechanism has to map vulnerabilities to devices, otherwise it may waste the resources of the prevention runtime. Indeed, it will look only after vulnerabilities which do target a device in the current SIP dialog. Therefore, we need to target preventions according to the existing devices in the controlled SIP network and their respective running state. This is enabled by the VeTo language.

IV. Language Overview

We describe here the main features of the VeTo language. Next, we discuss implementations issues of the language runtime within the SecSIP tool [11]. Finally, we present examples of specified prevention schemes from discovered vulnerabilities in real world SIP implementations. The full specification of the VeTo language can be found in [12].

VeTo relies on three features to counter a particular known vulnerability. The language combines a context, a definition and the event properties of a vulnerability to provide the ability to prevent against its exploitation. The underlying idea is to uncouple common from specific properties of vulnerabilities into reusable parts. The context block exhibits the vulnerability surrounding environment properties. The definition block provides the vulnerability related assumptions on its behaviour such as the involved SIP messages and their respective fields. The prevention block describes the vulnerable behaviour within its context and includes a response action. The definition and the context blocks can be shared by different vulnerabilities. However, each vulnerability has its own prevention block.

A. Context block

The context block describes the information associated to different specified protection blocks. The information includes the involved SIP devices, users and their surrounding environment such as time, locality, availability state, outbound proxy, etc. The context is used to trigger the proper protection block according to the current information available to the language runtime. A context is associated to one or several protection blocks, but a vulnerability has a single context block. In VeTo, a context is defined as a set of labeled attributes with predetermined values. Instead of a value, an attribute can have a set of values. The context attributes are mainly the URI of the targeted SIP entity, the time when a patch will be applied to remove the vulnerability and the firmware version where it is reported. Figure 1 shows a specification of a context block.

The `target` attribute denotes a SIP element that is concerned by a vulnerability protection scheme. A target value is composed of the transport protocol underlying the SIP traffic behind the vulnerability, an IP address, range or hostname of the targeted SIP element and the port of that SIP traffic. The `lifetime` attribute is specified as recommended in the RFC 3339. It denotes the date when the vulnerability will be fixed or removed. When this date is unknown, the attribute is omitted from the context block. The `include` property allows to include the content of another context. For example, consider

```
context GlobalCtx begin
target => udp:iphone.example.com:5060;
target => tcp:android.example.com:*;
target => *:netbook.example.com:*;
locality => high;
context end
context myCtx begin
include => GlobalContext;
lifetime => 2009-05-07;
context end
```

Fig. 1. A VeTo context specification

the context block called `GlobalCtx` that enumerates a list of sip devices using the target attribute. Then, there is a context called `myCtx` that includes `GlobalCtx` and contains a lifetime attribute to specify the patch time of a particular vulnerability. The `locality` attribute expresses how often a vulnerability may occur. It takes one of the following values: very low, low, medium, high and very high.

B. Definition block

The definition block relies on regular expression based pattern matching rules. Each rule is composed of an optional header part including a pattern matching statement and a mandatory action part. The header of a each rule is the composition of terms followed by the operator *@match* and a regular expression. The body of the rule is an action which defines typed variables to be used by a protection block. The term component refers to an element from the parsed tree of a SIP message. When the term value matches the given pattern the variable is created by the runtime.

Rule 1 shows a definition block named *contactDefs*. The block contains two rules. The first rule defines a *set* type variable used to store the values of the *contact* field of a SIP message. The contact field is referenced using the predefined constant *SIP:headers.contact*. The second rule contains a matching pattern against the *method* field of a SIP request. If the field matches the pattern INVITE, the action creates an event named *ev_Invite*.

Rule 1 A VeTo definition block

```
1 definition contactDefs begin
2 define:set[SIP:headers.contact] contacts;
3 when SIP:headers.method @match "^INVITE$" ->
4                              let:event ev_Invite;
5 definition end
```

C. Prevention block

The prevention block relies on event based rules which describe the logical part of the vulnerable behaviour. It specifies the event patterns and their respective actions when they are satisfied. The event pattern part uses the set of variables defined in the definition blocks. Each event block specifies a protection scheme and has a unique identifier, an optional context and a `uses` statement to use a set of definitions. The context of the protection scheme is specified using the @ symbol followed by the name of the context block. Rule 2 shows a protection block named `myProtection` with a single rule. The block has as context `myCtx` which was described earlier. The rule header part specifies an event pattern based on the event `ev_Invite`. The occurrence of the event triggers the `store` action against the previously defined set type variable `contacts`. The `store` action keeps in memory the collection contacts which contains the values of the contact field over observed SIP messages.

Rule 2 A VeTo prevention block specification

```
veto myProtection@{myCtx} uses ContactDefs begin
(ev_Invite) => store:contacts;
veto end
```

In VeTo, an event denotes the occurrence of something interesting under circumstances. These events are consumed by the event patterns of the protection block's rules. VeTo event patterns draw on event languages developed for active databases [4] and streams [23].

We formally define an event variable as $e_i(t)$ where at an instant t the event occurs. e_i specifies an event label and the instant t denotes the arrival time of a SIP message.

1) Events Sequence: The events sequence takes as input a list of n events labels. It specifies a particular order in which the events should occur. Formally, a sequence of events is defined as follows:

$$(e_1, e_2, ..., e_n) \equiv \exists t_1 < t_2 < ... < t_n, e_1(t_1) \wedge e_2(t_2) \wedge .... \wedge e_n(t_n) \quad (1)$$

The following example represents a simple pattern of two events labels *ev_ack* and *ev_invite*. The rule checks if the events *ev_ack* precedes the event *ev_invite*. If the pattern is matched, then the SIP message is dropped. We note that we

```
(ev_ack,ev_invite) -> drop;
```

can use short cut notion to describe events sequence. We can use the repetition operator ”*” as described below.

```
(ev_invite[*2],ev_200_OK,ev_ack)
```

2) Embedded events: An embedded event is an event that occurs at the same time that another event. It allows to match an arbitrary number of events that appear at same instant of the arrival of a SIP message. Formally, this pattern is defined as follows:

$$(e_1(e_2, e_3, ..., e_n)) \equiv \exists t_i, e_1(t_i) \wedge e_2(t_i) \wedge ... \wedge e_n(t_i) \quad (2)$$

3) Negation patterns: A negation pattern specifies an event that does not appear within a sequence of events. Formally, a negation pattern is defined as follows:

$$\begin{aligned}&(e_1, .., e_{j-1}, \sim e_j, e_{j+1}, ..., e_n) \equiv \\ &\exists t_1 < .. < t_{j-1} < t_{j+1} < ... < t_n, e_1(t_1) \wedge ... \wedge (e_n(t_n)) \\ &\wedge(\forall t_1 <= t_i <= t_n, \neg e_j(t_i))\end{aligned} \quad (3)$$

inria-00544976, version 1 - 9 Dec 2010

When the sequence is empty, the negation pattern specifies any event that is different from the specified event. The negation pattern is denoted by the symbol $\sim$. Rule 3 depicts the usage of a negation pattern. The definition block creates the variable `contacts` of a set type as depicted in line 2. It also creates an event named `Ev_BYE` as depicted in line 3. The rule in the Veto block checks the non occurrence of the event `Ev_BYE` to feed the contacts list with the values of the field `contact` of a SIP message.

Rule 3 An illustration example of the usage of a negation event pattern

```
1 definition NegationDefs begin
2 define:set[SIP:headers.contact] contacts;
3 when SIP:request.method @match "^BYE$" ->
4                           let:event Ev_BYE;
5 definition end
6
7 veto Negation uses NegationDefs begin
8 (~Ev_BYE ) -> store:contacts;
9 veto end
```

4) Temporal patterns: VeTo provides a time window operator to express a temporal relation between events within a sequence. The operator is used to check events over a time window. Its aim is to represent the fact that some statements are only true over a given period of time. The Digit operand is the time window value over which the pattern has to be matched. After this time window the pattern is invalid. For example, in Rule 4 the VeTo block prevents from a flooding attack where 1000 INVITE messages arrive in a one second time window.

Rule 4 The VeTo protection block against a flooding attack using temporal event patterns

```
definition SIPMessages begin
when SIP:request.method @match "^INVITE" ->
                        let:event ev_invite;
definition end

veto Flooding uses SIPMessages begin
([ev_invite[*1000],1]) -> drop;
veto end
```

D. Variables scope and extent

In *VeTo*, each variable has a scope and an extent. The scope determines where the variable and its value are associated. The extent determines when the value is associated to the variable at runtime. The scope of a VeTo variable is related to its definition block. A VeTo variable is visible over all protection blocks that references its definition block.

The extent of a VeTo variable is either message or dialog. A SIP dialog is defined using the triple (*Call ID, To Tag, From Tag*) fields of a SIP message. Typically, the predefined constants named as *SIP:request.**, *SIP:response.** and *SIP:headers.** have an extent over a message. They take a new value each time a new SIP message is observed and parsed. The variables defined into a definition block and used by a prevention block have an extent over a dialog. They have as running instances as existing SIP dialogs. A variable that takes its values over many dialogs is defined using the keyword `global` followed by its type and name.

E. Illustrative Examples

We illustrate the use of the VeTo language over well identified SIP vulnerabilities found in the literature. We have selected three types of vulnerabilities with different degrees of complexity. The vulnerability preventions described below share the definition block depicted in Rule 5.

1) Malformed messages: VeTo provides the capabilities to address malformed messages according to two approaches [8]. The first approach is the misuse prevention, where the rules describe a known discovered malformed pattern within a given SIP message. For example, it is reported on several VoIP security mailing lists [21] that many hardware SIP phones are sensitive to SIP messages with illegal size fields value. The protection block described in Rule 6 depicts a prevention against a vulnerable SIP message where the size of the date header is greater than an allowed size of 120 bytes. The size of the date field is computed using the symbol *&* that precedes the field name *SIP:headers.date*.

Rule 6 A prevention against a vulnerable *OPTIONS* message

```
veto IllegalSize uses SIPDefs begin
(ev_Options) -> if (&SIP:headers.date @ge "120")
{
          drop;
}
veto end
```

The second approach is a specification based description of SIP messages. In this approach a set of VeTo rules enforce the ABNF of SIP messages as have been specified in RFC 3261 [16]. The advantage of this approach is that it protects a SIP network from unknown or undiscovered malformed patterns that may crash deployed SIP implementations. However, using this approach, numerous false negative may occur since some implementations introduce their own specific SIP fields that do not exactly follow the SIP specification. Rule 7 depicts an example of an INVITE SIP message with a non compliant *Call-ID* header that will be dropped.

Rule 7 A prevention against a non compliant Call-ID SIP messages field

```
veto EnforceSpec uses SIPDefs,MalformedDefs begin
(ev_Invite(ev_Malformed)) -> drop;
veto end
```

2) Flooding attacks: In [5], the author was interested in SIP-specific DoS attacks by flooding SIP entities with SIP compliant messages. He proposes a detection method that relies on thresholds. The trivial one is a threshold on the number of INVITE messages allowed per dialog to a particular

inria-00544976, version 1 - 9 Dec 2010

Rule 5 Examples of VeTo definition blocks

```
definition SIPDefs begin
when SIP:request.method @match "^INVITE$" -> let:event ev_Invite;
when SIP:message.method @math "^OPTIONS$" -> let:event ev_Options;
when SIP:request.method @match "^ACK$" ->   let:event ev_Ack;
when SIP:request.method !@match "^$" -> let:event ev_Request;
when SIP:request.method @match "^REGISTER" -> let:event ev_Register;
when SIP:response.code @match "^200" -> let:event ev_Response;
definition end

definition MalformedDefs begin
when SIP:headers.Call-ID !@match "^\s*(Call-ID|i)\s*:\s*w+[@w+]$" -> let:event ev_Malformed;
definition end
```

URI. Rule 8 describes the prevention block specifying this type of threshold. Firstly, we track the target URI of each INVITE message using the action *store* that acts on the set type variable *targets*. We count the number of *INVITE* messages sent to each target from the collection using the action *apply* that acts on the count type variable *targets.count*. Finally, we check the number of observed INVITE against an allowed threshold. If the threshold value is crossed, we drop all subsequent INVITE messages.

Rule 8 Prevention from an INVITE flooding attack against a particular target URI

```
definition FlDefs begin
define:set[SIP:headers.uri.addr] targets;
define:counter(1,2000) targets.count;
definition end

veto FloodingTarget uses SIPDefs,FlDefs begin
(ev_Invite) -> {
        store:targets;
        apply:targets.count;
        if (targets.count @ge 10) {
            drop;
        }

}
veto end
```

The most interesting threshold is the upper bound of the number of allowed transactions per node. Firstly, we define a collection that tracks transactions per node. We also define a counter to track their number. Rule 9 describes the prevention block of this vulnerability.

Rule 9 Prevention from a transaction based flooding attack

```
definition FlDefs begin
define:set[SIP:headers.branch] transactions;
define:counter(1,60000) transactions.count;
definition end

veto FloodingTr uses FlDefs,SIPDefs begin
(ev_Request) -> {
         store:transactions;
         apply:transactions.count;
         if (transaction.count @ge "10") {
            drop;
         }
}
veto end
```

3) DoS using broken handshaking: This attack is based on broken SIP handshaking where the attacker sends an INVITE request and then ignores the 200 OK response refusing to send the ACK. The attacker proceeds with a large number of broken initiations in order to exhaust the target resources. Rule 10 shows the prevention block from this vulnerability. The first event rule in the veto block `VulHandShacking`

Rule 10 Protection block against the broken handshaking SIP attack

```
definition HandSDefs begin
define:set[SIP:headers.from] black_list;
definition end

veto VulHandShacking uses HandSDefs,SIPDefs begin
(ev_Invite,*,(ev_Response,~[ev_Ack,1])) -> {
     store:blackList;
}
(*) -> if (blackList @contains "SIP:headers.from") {
         drop;
}
veto end
```

defines an event pattern to detect a 200 OK response without an ACK message within a time window of one second. In such case, it feeds the set type variable `blackList` with the URI of the sources of the uncompleted handshake messages. This list is used by the second rule from the same VeTo block to disallow the attacker from sending more broken handshakes or SIP messages. The symbol * is a Kleen closure to denotes the occurrence of any event within the current dialog.

V. Evaluation and results

In this section, we present the evaluation of the VeTo language regarding its expressive power and the performance of its underlying prevention tool SecSIP.

A. Qualitative analysis

We have compared the complexity of VeTo, STATL and Snort languages regarding the number of line of code required to specify different vulnerabilities preventions described in section IV-E. The result of our analysis is depicted in Table I.

We note that we only counted the rules and the event block headers for VeTo specification. We observe that the Snort language is not suitable to prevent from all vulnerabilities since it is stateless as explained in section II. However, it is useful to

inria-00544976, version 1 - 9 Dec 2010

Vulnerability	STATL	Snort	VeTo
IllegalSize	8	1	3
EnforceSpec	7	1	2
FloodTarget	22	1	8
FloodTrans	22	-	8
VulHandShacking	21	-	7

TABLE I
SIZES (LOC) OF STATL, SNORT AND VETO SPECIFICATIONS FOR PREVENTING AGAINST DIFFERENT VULNERABILITIES

prevent from simple attacks like buffer overflow or flooding. Snort based preventions are limited since a signature specifies the attack occurrence rather than the vulnerable behaviour of the SIP protocol. The STATL language specification are up to 3 times longer than the corresponding VeTo specifications, because each vulnerability has its own state machine to be completely described.

B. Experiments

We have setup a testbed of 3 machines with a core 2 CPU cadenced at 2.93GHZ and 2 GB of RAM. The three hosts are connected through a 100 Mbits switched Ethernet. One host acts as a SIP proxy running a version 1.3.2 of an OpenSIPS server. The two others act as SIP user agents (UA). The SIP UAs are implemented using the *SIPp* [7] tool which allows to write customised SIP protocol interactions as XML documents. The prevention rules are applied on the SecSip tool [11]. The tool acts as a support runtime of the VeTo language and implements required features to intercept SIP message in real time and executes VeTo preventions to counter existing vulnerabilities. Our performance metrics are the number of established calls and their respective establishment delays. A call is established when the 5 SIP messages *INVITE*, *100*, *180*, *200* and *ACK* are delivered correctly between the two UAs.

1) Base capacity and overhead: Firstly, we have measured the performance of the SIP proxy and the UAs without the support of SecSIP. Secondly, we deployed SecSIP on the same host as the SIP proxy. We varied the number of call attempts between 1 and 1000 INVITE/second per a step of 100. SecSIP runs the prevention block *EnforceSpec* depicted in Rule 7.

Figure 2 shows the number of established calls between two user agents through an OpenSIPS SIP proxy with and without a deployed prevention runtime running VeTo rules. We observe that the base capacity for the SIP proxy without SecSIP is around 300 CPS. We enable SecSIP on the same host as the SIP proxy to intercept SIP traffic using the *nfqueue* interface and it runs the prevention block. The call handling capacity of the proxy is dropped to 165 CPS. The call capacity decrease is attributed to the SecSIP overhead.

Figure 3 depicts The call establishment delay which is measured on the caller (UAC) between the sending of the INVITE message and the receipt of its respective ACK. We observe a mean delay lower than 200 ms while the SIP traffic goes only through the SIP proxy. The delay becomes more important when we deploy SecSIP and the prevention block. It reaches 2 seconds under a number of concurrent call attempts

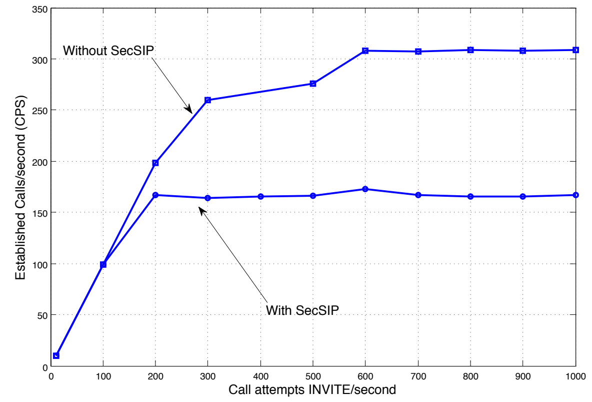

Fig. 2. Number of established calls under increasing number of calls in terms of INVITE/second, without and with a deployed SecSIP runtime

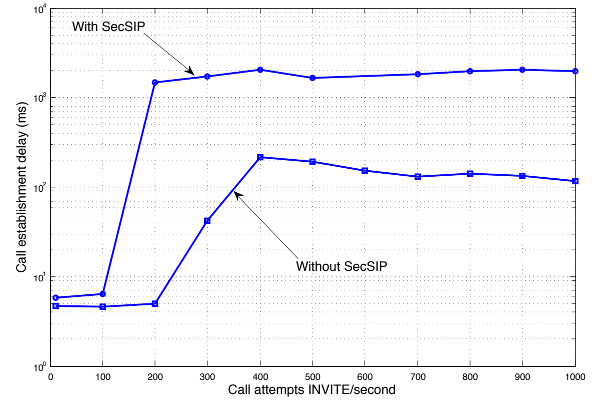

Fig. 3. Call establishment delay under increasing number of calls in terms of INVITE/second, with and without a deployed SecSIP runtime

around 200 CPS. Therefore, at worst a call needs 2 seconds to be established. The overhead introduced by SecSIP seems to be important but stays acceptable since the establishment delay is lower than the transaction expiration delay of 32 seconds. We have also verified that each INVITE and 200 OK messages and their respective responses 100 Trying and ACK messages fit within a time window lower than 500 ms.

2) Flooding prevention: In this scenario, we deployed on SecSIP the prevention block depicted in 8. This block prevents a source from flooding a target with high INVITE messages rate. The flow of INVITE messages originates from a single source with an increasing rate of 1 each 2 seconds. The starting rate is fixed at 1 INVITE/second. Figure 4 contains the measurements. We observe that the SecSIP runtime only allows a number of concurrent messages equal to 10. When an attack source exceeds this number and its traffic rate becomes high, then it is blocked and every incoming message from the attack source is dropped. The attack source has to wait a period of time before any of its incoming traffic will be forwarded.

VI. CONCLUSION AND FUTURE WORK

This work addresses the design of a description language, called VeTo for preventing the exploitation of SIP protocol

inria-00544976, version 1 - 9 Dec 2010

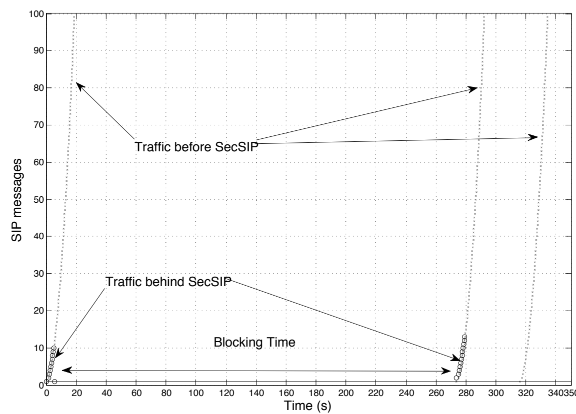

Fig. 4. Prevention against an INVITE messages flooding from a single source

vulnerabilities. The work is motivated by the important existing number of SIP vulnerabilities discovered using fuzzing processes or given in the literature. The VeTo language relies on a coupled rule event-based approach. A event approach is more practical in our context, since the vulnerable behaviour is expressed as a set of events towards a vulnerability point. The vulnerability point is the point where the protocol goes wrong. The events are the occurrence of something interesting under circumstances. The VeTo language is *stateful*. It includes instructions to record and maintain SIP protocol histories over messages, transactions and dialogs.

The language has been implemented into the SecSip tool which serves as its execution engine. Several vulnerabilities detection and avoidance schemes have been coded with the VeTo language. We have been interested in fuzzing-based discovered vulnerabilities. We have also specified several types of vulnerabilities behind flooding attacks. Veto specifications only addresses known vulnerabilities. Unknown vulnerabilities are not supported by our language since our aim was to protect a SIP network from discovered and unpatched vulnerabilities. Other techniques, like software testing and binary analysis may be used in conjunction to prevent from unknown and zero-day vulnerabilities. Currently, Veto rules are added manually to the SecSIP runtime when a new vulnerability is discovered. We are working on the automatic generation of these rules from fuzzing traces to provide complete and soundness rules. We are also working on the integration of the VeTo language and its execution engine into widely used SIP servers like Asterisk and OpenSIPS. We are also working on the integration of SecSip as a plugin for the Snort Tool to provide a more complete detection and avoidance scheme for SIP and its underlying protocols vulnerabilities.

References

[1] Humberto Abdelnur, Olivier Festor, and Radu State. KiF: A stateful sip fuzzer. In ACM, editor, *1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, July 2007.

[2] Nikita Borisov, David J. Brumley, and Helen J. Wang. A generic application-level protocol analyzer and its language. In *In 14th Annual Network & Distributed System Security Symposium*, 2007.

[3] Brian Caswell, Jay Beale, James C. Foster, and Jeremy Faircloth. *Snort 2.0 Intrusion Detection*. Syngress, May 2003.

[4] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[5] E.Y. Chen. Detecting dos attacks on sip systems. *1st IEEE Workshop on VoIP Management and Security*, pages 53–58, April 2006.

[6] Steven T. Eckmann, Giovanni Vigna, and Richard A. Kemmerer. STATL: an attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1-2):71–103, 2002.

[7] Richard Gayraud and Olivier Jacques. *SIPp Reference Manual*, Juin 2008. http://sipp.sourceforge.net/.

[8] Dimitris Geneiatakis, Georgios Kambourakis, Costas Lambrinoudakis, Tasos Dagiuklas, and Stefanos Gritzalis. A framework for protecting a sip-based infrastructure against malformed message attacks. *Comput. Netw.*, 51(10):2580–2593, 2007.

[9] Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21:181–199, 1995.

[10] Joseph Migga Kizza. *Guide to Computer Network Security*. Springer Publishing Company, Incorporated, 2008.

[11] Abdelkader Lahmadi and Olivier Festor. Secsip: A stateful firewall for sip-based networks. In *In the proccedings of 11th IFIP/IEEE International Symposium on Integrated Network Management, IM09, Long Island, New York, USA*, Juin 2009.

[12] Abdelkader Lahmadi and Olivier Festor. Veto: Reference manual. Technical report, Loria - INRIA Nancy Grand Est Research Center, July 2009.

[13] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[14] Gaston Ormazabal, Sarvesh Nagpal, Eilon Yardeni, and Henning Schulzrinne. Secure sip: A scalable prevention mechanism for dos attacks on sip based voip systems. In *Principles, Systems and Applications of IP Telecommunications. IPTComm 2008, Heidelberg, Germany, July 1-2, 2008. Revised Selected Papers*, pages 107–132, 2008.

[15] Eric Rescorla. Security holes... who cares? In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association.

[16] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916.

[17] D. Schwartz and J. Barkan. End-to-end route management in the session initiation protocol. http://tools.ietf.org/html/draft-schwartz-sip-routing-managment-00, February 2006.

[18] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.*, 38(4):207–218, 2008.

[19] R. Sparks, S. Lawrence, A. Hawrylyshen, and B. Campen. Addressing an Amplification Vulnerability in Session Initiation Protocol (SIP) Forking Proxies. RFC 5393 (Proposed Standard), December 2008.

[20] Giovanni Vigna, Steven Eckmann, and Richard Kemmerer. Attack languages. In *In Proceedings of the IEEE Information Survivability Workshop*, 2000.

[21] VoIPSA.org. VOIPSEC mailing list on VoIP security issues. http://voipsa.org/mailman/listinfo/voipsec_voipsa.org, January 2009.

[22] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Comput. Commun. Rev.*, 34(4):193–204, 2004.

[23] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.

[24] Ruishan Zhang, Xinyuan Wang, Xiaohui Yang, and Xuxian Jiang. Billing attacks on sip-based voip systems. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.

inria-00544976, version 1 - 9 Dec 2010