HAL
open science

VeTo: reference manual

Abdelkader Lahmadi, Olivier Festor

To cite this version:

Abdelkader Lahmadi, Olivier Festor. VeTo: reference manual. [Research Report] RT-7816, 2011, pp.51. hal-00645913

HAL Id: hal-00645913
https://inria.hal.science/hal-00645913

Submitted on 28 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

VeTo: reference manual

Abdelkader Lahmadi — Olivier Festor

N° 7816

February 2011

Thème COM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--7816--FR+ENG

VeTo: reference manual

Abdelkader Lahmadi , Olivier Festor

Thème COM — Systèmes communicants
Équipes-Projets Madynes

Rapport technique n° 7816 — February 2011 — 48 pages

Abstract: The SIP protocol is established as the defacto standard for media session signaling, in particular for voice-over IP services. Many research works and alert bulletins have reported various vulnerabilities in this protocol. These vulnerabilities are either inherent to the protocol specification or arise as flaws within SIP stack implementations or erroneous configurations. To protect SIP-based networks from the exploitation of such vulnerabilities, patches may be released for the implementation bugs, the SIP specification may be revisited to cover the specification errors and configuration guidelines can be issued to offer good configuration receipts to administrators. The time to patching and revisiting specification may be considerable. To overcome this problem, a first-line of defense against SIP vulnerabilities has to be developed. In [2], we have presented a stateful firewall architecture dedicated to SIP-based networks protection. The firewall runtime uses a domain specific language, called VeTo. Its design, syntax and semantics are described in this work.

Key-words: VoIP, SIP, vulnerabilities, language, protection, VeTo

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

VeTo : manuel de référence

Résumé : Le protocole SIP est aujourd'hui le standard de fait pour la signalisation des sessions multimédia à l'échelle de l'Internet. Plusieurs travaux ainsi que des bulletins d'alertes ont reporté l'existence des différentes vulnérabilités au niveau de ses implantations, de ses spécifications, de ses implémentations et de ses paramétrages. La protection du protocole SIP de l'exploitation de ces vulnérabilités nécessite l'application des patches au niveau de ses implantations à bien que la révision des ses spécifications et la publication de recettes de bonnes pratiques pour sa configuration. Ces actions prennent un temps considérable avant d'être menées. Afin de résoudre ces problèmes, une premire ligne de défense nécessite d'être mise en place. Dans [2], nous avons proposé une architecture de défense reposant sur un pare-feu dédié au protocole SIP. Ce pare-feu s'appuie sur un langage, nommé VeTo dédié à la spécification de règles de prévention contre les vulnérabilités présentes dans le protocole SIP. Ce rapport détaille la syntaxe, la sémantique et son infrastructure support.

Mots-clés : VoIP, SIP, vulnérabilités, language, protection, VeTo

Contents

1 Introduction **5**
1.1 Introduction . . . 5
1.2 Overview of SIP-based services . . . 6
1.3 Taxonomy of SIP specific vulnerabilities . . . 7
1.3.1 SIP messages . . . 7
1.3.2 SIP routing . . . 8
1.3.3 SIP authentication . . . 8

2 VeTo Syntax and Semantics **11**
2.1 The VeTo Language Syntax . . . 12
2.1.1 Lexical conventions . . . 12
2.1.2 Lexical syntax . . . 12
2.2 VeTo blocks and contexts . . . 13
2.2.1 Context block . . . 13
2.2.2 Definition block . . . 14
2.2.3 Event patterns block . . . 15
2.3 VeTo Variables . . . 16
2.3.1 Predefined variables . . . 16
2.3.2 Built-in functions . . . 16
2.3.3 Collections . . . 16
2.3.4 Event variables . . . 18
2.3.5 Event patterns . . . 19
2.3.6 More event patterns examples . . . 22
2.3.7 State variables . . . 22
2.4 Variables scope and extent . . . 23
2.5 VeTo Conditions . . . 24
2.5.1 Definition of conditions . . . 24
2.5.2 Semantics of conditions . . . 24
2.5.3 Examples of Conditions . . . 24
2.6 VeTo actions . . . 25
2.6.1 *LET* action . . . 25
2.6.2 *STORE* action . . . 25
2.6.3 *ASSIGN* action . . . 25
2.6.4 *APPLY* Action . . . 25
2.6.5 *DROP* action . . . 26
2.7 Implementation issues . . . 26

2.7.1 Rules compilation and checking . . . 26
2.7.2 Events pattern matching . . . 28

3 VeTo Testing **31**
3.1 Malformed messages vulnerabilities . . . 31
3.1.1 Incorrect Grammar Message . . . 31
3.1.2 Cross-site scripting attacks over SIP . . . 32
3.2 Invalid semantics . . . 33
3.2.1 Out of state message . . . 34
3.3 Flooding attacks . . . 34
3.4 Implementation flaws specific attacks . . . 34
3.5 Legitimate SIP messages based vulnerabilities . . . 35
3.5.1 SIP Method based attacks . . . 36
3.5.2 The SIP DNS flooding attack . . . 37
3.5.3 Ringing-based DoS attack . . . 38
3.5.4 The SIP Identity baiting attack . . . 39
3.5.5 The Misuse-ringing attack . . . 39
3.5.6 The re-INVITE message syndrome . . . 40

4 Conclusion **45**

Chapter 1

Introduction

1.1 Introduction

VoIP networks is a generic term that covers deployments ranging in complexity from hobbyists using the internet to get free phone calls on a peer to peer basis, to full scale PSTN replacement networks [10]. However, before the achievement of a full transition from a PSTN scale solution to a full VoIP network, many issues have to be addressed. Among them, the most important issues are:

- Signaling Protocol,
- Quality of Service,
- Reliability and Availability,
- Security.

Different Signaling Protocols have been designed for VoIP networks. They include device control protocols such as H.248 (Megaco), MGCP, NCS, etc; Access services signaling protocols such as SIP, H.323, etc and network service signaling protocols such as SIP, SIP-T, BICC, CMSS [13]. Over the years, the SIP protocol [20] has established itself as the defacto standard of a network service signaling protocol for VoIP networks.

One of the key requirement of a VoIP service is its call quality. It has to be at least equivalent to the existing PSTN. The quality is mainly related to the delay, jitter and packet loss. These performance criteria have to be kept low to guarantee a suitable QoS. Therefore, when special processing is added to a VoIP traffic it should have a low overhead and keep the above performance metrics stable. Therefore additional security solutions have to solve the problems in real time [8].

Another issue when carrying VoIP service is its reliability and availability. A VoIP network should achieve at least five nines reliability, equivalent to fewer than five minutes per year downtime.

VoIP is an added value service, therefore, it needs a strong security model to guarantee its reliability and availability. The main challenge with VoIP security is the nature of its running environment and technologies since it relies on open technologies like SIP using servers reachable through the Internet. Such technologies are implemented in software and provided often over general-purpose computing hardware. Therefore, VoIP can suffer from similar security threats as HTTP and SMTP based services. Currently, VoIP is in a rapid evolution phase deployments by service providers, with a large number of new services. For example, instant messaging, video conferencing and integration of VoIP with other business systems including E-mail, Customer Relationship Management (CRM), and Web systems. We have counted **64** RFCs only related to the SIP protocol. The consequence of this complexity is that many vulnerabilities

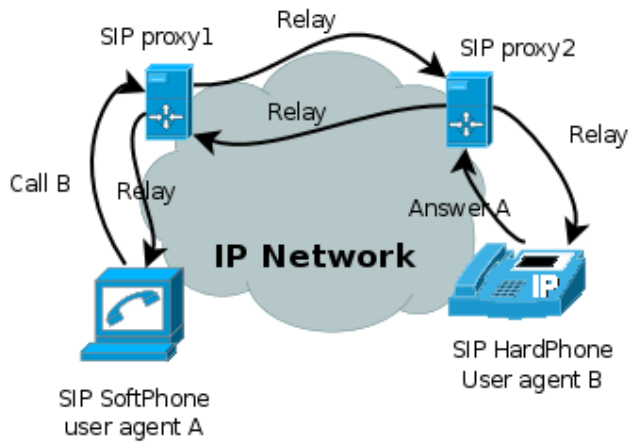

Figure 1.1: An example of SIP infrastructure for voice services.

are undiscovered, some within the SIP protocol specification and many within its implementations. These vulnerabilities may be used to gain access to or disturb the delivery of SIP based services. The SIP protocol becomes attractive to attackers due to its HTTP underpinning and the ASCII format of its packets. Since SIP is used as a converged protocol, it inherits all the security weaknesses from other underlying protocols like spoofing, sniffing, denial of service, spamming, vishing,... [18].

1.2 Overview of SIP-based services

SIP services like VoIP are currently dominated by the use of the SIP protocol [20] as a signaling protocol. It is an application-layer protocol that relies on a request-response model. The main functions of SIP within a VoIP service is the establishment, the modification and the termination of calling sessions between networked SIP elements.

As depicted in Figure 1.1, the SIP protocol relies on the open Internet architecture to relay its messages between its different SIP-based networked elements. A SIP network typically comprises the following entities:

- User Agent (UA) is a logical function in the SIP network that initiates or responds to SIP messages. It may act as a UA client or a UA server. The UA client initiates requests and accepts SIP responses. The UA server accepts requests and sends responses.

- SIP proxy is an intermediate element within a SIP network that is responsible for routing SIP messages to UA or other proxies.

- Registrar is a server that accepts REGISTER requests and keeps the information it receives for further callee looking-up.

The SIP protocol uses messages to establish sessions. The messages grouped together to establish a session form a dialog. Each dialog is identified by the call identifier (*Call-ID*), a local tag and a remote tag fields in a SIP message. A dialog consists of a series of transactions between the UAs. Each transaction consists of a single request and any responses to that request. Transactions are identified within a dialog using the *branch* field and are defined across dialogs using the *Cseq* field.

Type of Threat	Threats
Denial of Service	Request flooding Malformed requests and messages Faked messages injection Call Hijacking
Theft of service	Unauthorized billing
Misuse of service	Unwanted contact Unwanted content (SPIT) Billing adjustment (maximize or minimize)

Table 1.1: List of SIP specific threats.

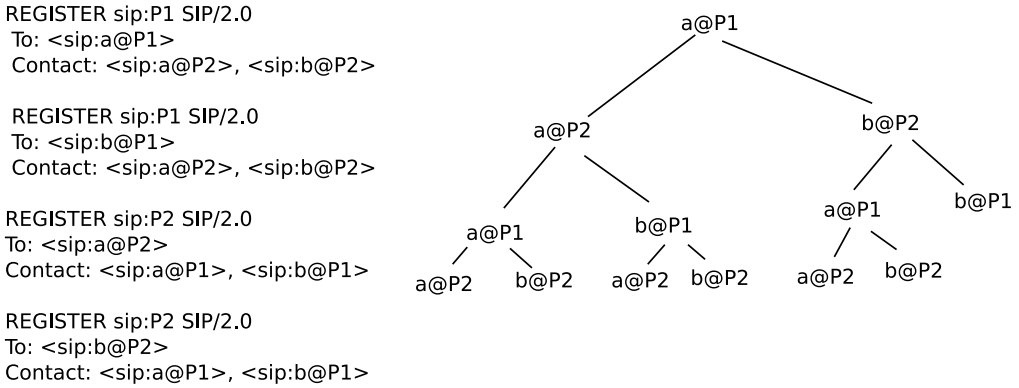

Figure 1.2: INVITE message propagation using the forking-proxies vulnerability.

1.3 Taxonomy of SIP specific vulnerabilities

In [24], authors provide a comprehensive list of VoIP threats. Rather than repeating their results, we list SIP-specific threats. The table 1.1 lists those threats related to a SIP based service. These threats are due to the weakness that shows the SIP protocol on its different inherent features. We focus our analysis on the SIP related weakness disregarding those due to underlying protocols.

1.3.1 SIP messages

SIP messages are text-based and use the UTF-8 charset. This feature leads to an easy modification of them by man in the middle attackers. The SIP protocol uses the *INVITE* message to setup and modify dialogs over their lifetime. According to RFC 3261, the message used to modify dialogs is called re-INVITE, but it has the same method value as INVITE. This may results to a theft of service by exploiting a spoofed or simply relaying a crafted re-INVITE message to initialize 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 UAS. Recently, the RFC 5393 [23] proves that this feature may causes a massive flooding attack with valid SIP requests between proxy-to-proxy messages. A version of this attack demonstrates that fewer than ten messages can lead to potentially 2^{71} attack messages. An instance of this attack is depicted in figure 1.2.

1.3.2 SIP routing

The SIP protocol needs some mechanism to determine which path a message needs to take to reach a destination. SIP embeds packet routing information into headers. There are four types of routing headers: record route headers, route headers, via headers, and contact headers.

Routing headers

The *record-route* header is added by a proxy to tell the recipient that he want to remain in the signaling path for all future SIP requests within the dialog.

Using the *record-route* header, SIP elements compiles a list of the IP addresses or fully qualified domain names between the source host and the destination host. This list essentially acts as a route that the message must take in order to reach its destination, with each entry on the list serving as a hop along the route. When a host receives a message, it checks to see whether or not it is the final recipient. If not, the host removes its own IP address or fully qualified domain name from the route header and then forwards the message to the next host on the list. The process is repeated until the packet reaches its destination.

Via headers work similarly to route headers, except that they work in the opposite way. Suppose, for instance, that a source host and a destination host are geographically dispersed, and that SIP traffic must flow between multiple proxies in route to the destination. In such a situation, each of the proxies that the message passes through adds its own IP address or fully qualified domain name to the via header. The idea is that when the message arrives at its destination, the destination host can determine exactly what path the message took to reach its destination, and which hosts have handled the message along the way. In some instances, this information is also used to create a return path. The *contact* header lets the recipient proxy determine to which user the message should be directed.

Routing weakness

SIP routing relies on the proxy elements that take downstream routing decisions based on the routing headers and upstream routing decision based on Vias fields. This routing mechanism leverages the following weaknesses [22]:

- Any element can insert/delete/alter routing headers.
- Proxies route statelessly without call-route state or global route knowledge.
- There is no authoritative, trusted element for end-to-end routing policy.
- SIP privacy directives allow for stripping of route information.

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

1.3.3 SIP authentication

As described in [20], SIP mandates the use of HTTP digest-based authentication as a security mechanism to protect SIP messages. It provides an anti-replay protection and one-way authentication to SIP messages. Existing SIP authentication has the following weakness [28].

- It is not an end-to-end security model. It is difficult to protect SIP messages on an end-to-end basis since SIP servers and proxies need to examine and change certain fields.

- It only applies to a few SIP messages (INVITE, BYE, REGISTER), and leaves other important SIP messages (TRYING, 200 OK, ACK, BUSY) unprotected.
- It only protects a few SIP fields (Request-URI, realm), and it leaves other important fields unprotected (e.g, From, To).
- It only applies to SIP messages from the UAC to SIP servers, and it leaves all the SIP messages from the SIP servers to UAC unprotected.

These vulnerabilities mainly generate billing attacks as described in [28].

Chapter 2

VeTo Syntax and Semantics

The VeTo language relies on a mixture of rule based programming and pattern matching models. The rule based programming style is well adopted for flexible data manipulation and for specifying IF-THEN logic. The pattern matching model provides facilities to manipulate data, to define events and to check SIP protocol properties involved in a vulnerability prevention. The VeTo language relies on a negative logic which means it describes exceptions rather than the traditional *allow* rules in standard firewall languages. While designing the VeTo language we have fixed the following features:

- Basic variable types for specifying particular derived variables other than SIP protocol structural data;
- Pattern operators for expressing relationship between SIP protocol data and patterns;
- Sateful variables to store their values in different types of containers;
- Regular expression patterns that allow to express patterns that match SIP protocol fields values;
- Event patterns that allow to express the set of events and sub-events involved in a SIP vulnerability;
- Condition that lets us restrict matches of events to specific pattern;
- Logical operators that allow us to specific logical relations between matched patterns.

To provide these features, the VeTo languages relies on the three types of blocks: definition, context and prevention. They are mapped into the three properties of a vulnerability which are its input SIP protocol data units, its running context and its logical behaviour. Therefore, a prevention against a vulnerability, requires to capture the vulnerability logic using behavioural rules and the context of this behaviour. The definition part manages the underlying structural input data of the SIP protocol. It provides information from the parsed SIP messages involved in a vulnerability. The prevention part interprets the logic part of the vulnerability and takes actions against the vulnerability exploitation. The context part provides sensitive information about the vulnerability environment. These three concepts are alike the traditional MVC models [19] in software engineering.

In the following sections, we describe the syntax of the VeTo language variables, patterns, operators, condition and actions.

2.1 The VeTo Language Syntax

The syntax of the VeTo language is described using the EBNF[1] notation. In the VeTo syntax, a terminal symbol is enclosed in single quotes. Optional items are enclosed in square brackets [ ]. An element with a trailing operator "*" denotes an item that may appear zero or more times.

2.1.1 Lexical conventions

We used the following conventions to define the syntax of the VeTo language.

```
Value::=Number|Identifier
Number::=1*Digit
Identifier::=Letter(Letter | Digit| '_' | '-')*
Digit::='0' ... '9'
Letter::='a' ... 'z' | 'A' ... 'Z'
```

VeTo variables names use the following conventions.

```
Term::='SIP:' Type '.' Field
Type::= 'request' | 'response' | 'message'
Field::= 'method' | 'status-code' | 'uri' | 'Call-ID'
VariableName::= Identifier
EventName::=Identifier
StateName::=Identifier
CounterName::=Identifier
TimerName::=Identifier
CollectionName::=Identifier
BlockName::=Identifier
Parameters::= Value | Value ',' Parameters
```

The complete list of terminal values of the symbols *Type* and *Field* is presented in the Annex **??**.

2.1.2 Lexical syntax

The VeTo language provides three types of blocks, where each block is either a definition, a context or an events handler. The definition block only contains pattern matching rules based on regular expressions over SIP messages fields. Each rule contains a matching pattern and an action part that generates an event when the pattern is satisfied. The event block uses the definition blocks and contains a set of rules with event based patterns. Each rule contains an events pattern, a set of conditions which constraint the events, and a set of actions to be taken when the pattern is satisfied. The events referenced by these rules are those defined in the definition block. The protection against a vulnerability exploitation is described by the events patterns block where actions are executed to prevent the exploit of the vulnerability on a particular device.

The prevention block delimiter is the keyword '*veto*' followed by the identifier of the block, the identifier of the context of the rules, the set of definition blocks identifiers and the keyword 'begin'. The block ends with the keywords 'veto' followed by 'end'. The context block rules contains mainly the devices targeted by the vulnerability. The block of rules is only applied to these devices. The wild card character "*" represents all devices existing in a target SIP network.

A Veto rule may specify or not a condition. An action is executed, if the condition is satisfied. If the condition is not specified, the action is executed once within each rules processing cycle. A Veto condition is

[1] EBNF: the Extended Backus-Naur form is a meta-syntax notation used to express context-free grammars.

a composition of a set of assertions related by the operators && and |||| that denote respectively conjunction and disjunction. An assertion is composed of a Veto variable, an operator and a pattern. Several built-in operators are provided for comparing and containing operations. The comparing operators include equality (*@eq*), greater (*@ge*) or less (*@le*). The operators to be applied on collections are (*@contains*, *@in*). These operators check if a collection contains or not a particular value.

2.2 VeTo blocks and contexts

A protection against a vulnerability exploitation is described using the three types of blocks: context, definition and event patterns handlers. Vulnerabilities might share the context and definition block, but each of which has its own prevention block that describes its events patterns and the action to be taken against its exploitation.

2.2.1 Context block

A context block describes information associated to one or several protection blocks. This information is used by the VeTo runtime to trigger the proper protection block. It is used to select the different protection blocks which apply to the current context. Context information include the existing SIP devices, users and their surrounding environment such as time, locality, availability state, outbound proxy, etc. The syntax of a context block is as follows:

```
ContextBlockBegin::='context' ContextID 'begin'
ContextID::= BlockName
ContextBlockEnd::='context' 'end'
ContextBlock::=ContextBlockBegin ContextProperties ContextBlockEnd
ContextProperties::= ContextProperty ';' | ContextProperty ';'
                     ContextProperties
ContextProperty::= PropertyName '=>' Property  ';'
PropertyName::= 'target' | 'include' | 'locality'
Property::=TargetProperty | IncludeProperty | LocalityProperty
TargetProperty ::= TargetURI [',' TimeSpec ',' Version]
TargetURI ::= Protocol ':' (IPaddr | IPrange | HostName) ':' Port
Protocol::= 'udp' | 'tcp' | 'sip'| '*'
TimeSpec::=  date-fullyear "-" date-month "-" date-mday ;
date-fullyear   ::= 4DIGIT
date-month      ::= 2DIGIT  ; 01-12
date-mday       ::= 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on month/year
Version ::= 1DIGIT | 1DIGIT '.' Version
IncludeProperty ::= ['{'] IncludeID ['}']
IncludeID ::= ContextID | ContextID ',' IncludeID
LocalityProperty ::= 'low' | 'high' | 'medium' | 'very low' | 'very high'
Values::= Value | Values ',' Value
```

A context is specific to one or multiple vulnerability blocks. On the opposite, a vulnerability has a single context. 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 of validity (duration) of the protection, when known and the firmware version where the vulnerability is reported. The *target* attribute denotes a SIP element that is related to the described

vulnerability. A target value is composed of the transport protocol underlying the SIP traffic behind the vulnerability, an IP address, range or the hostname of the targeted SIP element, the port of that SIP traffic. and a lifetime value which is specified as recommended in RFC 3339. The lifetime value denotes the date when the vulnerability will be fixed or removed. When this date is unknown, this value is omitted from the context block. The *include* property allows to include the contents of another context. For example, consider a context called "SIP_PROXIES" that enumerates a list of sip proxies using the target attribute. Then, there is a context called "SIP_SERVERS" that includes SIP_PROXIES and contains a list of SIP servers. The *locality* attribute expresses how often a vulnerability is likely to be exploited. It takes a value among: *very low, low, medium, high and very high.* The following example illustrates the different attributes of a VeTo context.

```
CONTEXT context_example BEGIN
TARGET => udp:myiphone.mydomain.com:5060, 2009-05-07 ;
TARGET => tcp:myhtcmagic.mydomain.com:* ;
TARGET => *:mylaptop.mydomain.com:*, 2009-05-10 ;
LOCALITY => high;
CONTEXT END
CONTEXT context_global BEGIN
INCLUDE => context_example;
CONTEXT context_global END
```

2.2.2 Definition block

The definition block carries the different variables to be used by the prevention blocks. It relies on regular expression pattern matching rules against SIP message fields. The header of a rule is the composition of a message term followed by the operator *@match* and a regular expression. The body of the rule is a set of instructions to define variables constructors. A message term refers to an element from the parsed tree of a SIP message. When a given message term matches the pattern, the defined variables are instantiated by the runtime. A definition block has the following syntax:

```
DefinitionBlockBegin::='definition' DefinitionBlockName 'begin'
DefinitionBlockEnd::='definition' 'end'
DefinitionBlock::=DefinitionBlockBegin DefinitionRules DefinitionBlockEnd
DefinitionRules::= DefinitionRule ';' | DefinitionRule ';' DefinitionRules
```

A definition rule is defined as follows:

```
DefinitionRule::= ['when' Term [!] '@match' Pattern '->'] PatternActions
PatternActions::=   PatternAction ';' | PatternAction ';' PatternActions
PatternAction::= 'Let:' (EventConstructor|StateConstructor|
   CounterConstructor|TimerConstructor|
                  CollectionrConstructor)
EventConstructor::='event' EventName;
StateConstructor::='state' StateName
CounterConstructor::='counter' '(' Parameters ')' CounterName
TimerConstructor::='timer' '(' Parameters ')' TimerName
CollectionConstructor::=CollectionType '['Term']' CollectionName
CollectionType::='set' | 'list' | 'bag'
```

The VeTo language includes several built-in variable types: *event*, *state*, *counter*, *timer* and *collections*. The collections of type *set*, *list* and *bag* are used to store messages term values.

The Rule 1 listing depicts a definition block named *contactDefs*. The block contains two rules presented in lines 2 and 3. The first rule defines a variable of type collection which stores the values of the contact field of the current SIP message. The second rule contains a matching part against the message term *sip:request.method* which references the method field of a SIP request. If the term value from the current message matches the pattern INVITE, an event named *ev_Invite* is created from the current dialog.

Rule 1 An example of a definition block.

```
1 DEFINITION contactDefs BEGIN
2 LET:Set[sip:message.contact] contacts;
3 WHEN sip:request.method @MATCH "^INVITE$" -> LET:EEVENT ev_Invite;
4 DEFINITION END
```

2.2.3 Event patterns block

The event block specifies event based patterns involved in a vulnerability and their respective actions when the patterns are satisfied. It represents the logic part of the vulnerability. The event pattern part uses the variables defined in the definition block. The event block has a unique identifier, a context and uses a set of definitions. The main syntax of a VeTo event block is as follows:

```
1 VeToBlockBegin::= 'veto' BlockName'@'{'ContextID'}' uses
     DefinitionBlockName 'begin'
2 VeToBlockEnd::= 'veto' 'end'
3 Definitions::= Identifier | Identifier ',' Definitions
4 VeToRules::= VeToRule ';' | VeToRule ';' VeToRules
5 VeToBlock::= VeToBlockBegin VeToRules VeToBlockEnd
```

The *contextID* of a VeTo block denotes the identifier of a context block. This restricts the set of rules to a particular context defined in a context block. An event based rule has the following syntax:

```
1 VeToRule::= EventPattern '=>' Body
2 Body::= [Conditions] '{' Actions '}' ';'
3 Conditions::= 'if' '(' Condition ')'
4 Actions::= Action ';' | Action ';' Actions
5 Condition::= Assertion * (LogicalOperator Assertion)
6 LogicalOperator::='||' | '&&'
7 Assertion::= ['!']Variable'.'Operation'(' Value ')'
8 Variable::=['&'] VariableName * ('.' VariableName)
9 Operation::= 'eq'|'ge'|'contains'|'in'
10 Action::= DistriputiveAction | NonDistriputiveAction
11 DistriputiveAction::= 'drop'
12 NonDistriputiveAction::=  StoreAction |AssignAction
13 StoreAction::='store' ':' CollectionName
14 AssignAction::='assign' ':' VariableName '=' Value
```

The body part of an event rule contains a mandatory set of actions. The rule is activated if the event pattern is satisfied. The following example shows an event block which contains a single rule. The rule header part

checks an event pattern based on the event *ev_Invite*. If the event occurs, then the rule is triggered and the action *store* is executed.

```
VETO InviteContacts@{} USES ContactDefs BEGIN
(ev_Invite) => STORE:contacts;
VETO END
```

2.3 VeTo Variables

The Veto language provides two classes of variables: stateful and stateless. The main difference between them is their persistence over SIP messages. A stateless variable does not retain its value over SIP messages. It changes its value from one message to the next. A stateful variable (or collection) has some internal storage that allows it to hold its value over the processing of multiple SIP messages.

2.3.1 Predefined variables

In the Veto language predefined variables are provided by the implementation. They have an extent over the current SIP message lifetime. By default the *SecSip* runtime provides a set of predefined variables mapped to the parsed fields of a SIP message. These variables are identified by *SIP* messages field identifiers. We note that these predefined variables are stateless. For example, the predefined variable *sip:headers.to.addr* takes its value from the field *To* of a SIP message. The following rules use the predefined variable *sip:request.method* in their respective left parts.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "^INVITE$" -> LET:EVENT Ev_Invite;
WHEN sip:request.method" @MATCH "^ACK$" -> LET:EVENT Ev_Ack;
DEFINITION END
```

2.3.2 Built-in functions

The VeTo language provides some buit-in functions to be applied to its arguments in the action part of a triggered event rule. The main built-in function is the counter. The counter is a function with incremental and decremental behaviors. It behaves like a gauge or a counter according to the number of provided parameters. If no parameter is provided, the function behaves as an increasing counter and it is incremented when an event pattern is matched. When a decreasing value and an interval are given as parameters to the counter function, then after each time interval, it is decremented by the specific value. Rule 2 defines a variable of type counter which is incremented every time an INVITE message is observed and it is decremented by 10 every 60000 milliseconds.

2.3.3 Collections

A collection variable stores the values of a predefined variable of the current SIP message. Its values are collected within a container over a dialog. The syntax to store the values of a variable is as follows:

Rule 2 An illustration example of the usage of the built-in counter function.

```
DEFINITION CounterDefs BEGIN
LET : COUNTER(10,60000) count;
WHEN sip:request.method @MATCH "^INVITE$" -> LET:EVENT ev_Invite;
DEFINITION END

VETO CountInvite USES CounterDefs BEGIN
(ev_Invite) -> APPLY:count;
VETO END
```

```
'let' ':' ContainerType '['VariableName']' CollectionName
'store' ':' CollectionName
```

Firstly, the action *let* creates an instance of the variable *CollectionName* of type identified by *ContainerType*. It also associates the predefined variable that owns the values to the collection variable. The *VariableName* denotes a statless variable which may be either a SIP message field or a user-defined variable The *ContainerType* specifies the type of the container where the collection values are stored. The VeTo language provides three types of containers to store collections: sets, lists and bags. A set is an unordered collection of objects without repeated values. It may be used to keep a single value over a SIP dialog. A list is an ordered collection of object values. A bag is a set that allows repeated values from multiple objects.

After being initialized, the *store* action starts recording the values of the variable *VariableName* into the collection variable instance. In Rule 3, we define a collection variable named callidList that stores the value of the call-ID field from each observed *INVITE* SIP message within a dialog.

Rule 3 An illustration of the usage of the collection variables.

```
DEFINITION CollectionDefs BEGIN
LET: SET[sip:request.callid] callidList;
WHEN sip:request.method @MATCH "^INVITE$" ->  LET:EVENT ev_Invite;
DEFINITION END

VETO collection USES CollectionDefs BEGIN
(ev_Invite) -> STORE:CallidList;
VETO END
```

Collections keys

The different collection instances created over SIP dialogs or transactions are stored in a hash table. Where each collection instance has a key that is either the dialog or the transaction identifier. The dialog identifier is defined by *<Call ID, To Tag, From Tag>* and a transaction identifier is defined by *<branch value, Cseq method>*.

The VeTo language provides a powerful feature that relies on the dot notation to define the keys of collections. To illustrate this feature, we take the following example. Suppose that we need to record the number of *INVITE* messages addressed to each different SIP device in our network. A SIP device is defined by its IP address embedded within the *Request-URI* field of the *INVITE* message. We need a collection variable that stores the different SIP devices addresses. We also need a variable that counts the number of

Invite messages. The number of INVITE messages has an addressing space over the recorded URI from the INVITE requests. Herein, we need to map the number of INVITE messages over SIP devices URIs. Thus, we use a *set* as a type for the stateful container of the SIP devices addresses. The values of this set, has to be used as a key to group the number of Invite message values associated per SIP device. This mapping between the collection variable values and a stateless variable is done using the variable composition, which is denoted by the infix operator '.'. Let S a collection variable and O is a stateless one, where S is the inner variable and O is the outer one. The VeTo expression $S.O$ means that *Key(O)=Value(S)*, and *Key(S)= DialogID*. Where *DialogID* is the dialog identifier of the observed SIP messages. The following listing shows a set of VeTo rules to describe the number of *INVITE* messages collected using the observed *SIP URIs*.

```
DEFINTION CounterDefs BEGIN
LET: SET[sip:request.uri.addr] uris;
LET: COUNTER(10,60000) uris.rate;
WHEN sip:request.method "^INVITE" -> LET:EVENT ev_Invite;
DEFINITION END

VETO CounterExample USES counterDefs BEGIN
(ev_Invite){
        STORE:uris;
        APPLY:uris.rate;
}
VETO END
```

The type *COUNTER* is a built-in function provided by the VeTo language. The two optional arguments 10 and 60 mean that the counter has to be decreased by 10 each 60 000 milli-seconds. The action *APPLY* applies the counter function to arguments.

2.3.4 Event variables

The event type defines variables that represent the occurrence of something interesting under circumstances. For example, a regular expression pattern is matched over a SIP message field within a specific dialog. An event variable is defined using the *LET* action followed by the built-in *EVENT* type and its identifier. The instance of the event variable is created within a dialog. In the following example, we define different event variables using the matching operator against different fields of observed SIP messages. The event variables are defined and created within a definition block. The named event variables like *ev_ack* and *ev_Invite* are used by Veto event blocks to compose event patterns and take actions.

```
DEFINITION SIPMessages BEGIN
WHEN SIP:request.method @MATCH "^ACK$" -> LET:EVENT ev_ack;
WHEN SIP:request.method @match "^INVITE" -> LET:EVENT ev_invite;
DEFINITION END
```

2.3.5 Event patterns

An event pattern is a template that matches a list of variables of type event. It recognize defined events causalities, dependencies and timing. We note that each event variable within a pattern should be created using the action *let* before being referenced by an event pattern. The syntax of an event pattern is as follows:

```
EventPattern ::= '(' Sequence ')'
Sequence ::= Event | Event ',' Sequence

Event ::= InstantEvent | RepeatedEvent | TemporalEvent

InstantEvent ::= SingleEvent  | SingleEvent '(' EmbeddedEvents ')' | AnyEvent
EmbeddedEvents ::= SingleEvent | SingleEvent ',' EmbeddedEvents
SingleEvent ::= [ '~' ] Label
AnyEvent ::= '*'

TemporalEvent ::= [ '~' ] '[' RepeatedEvent ',' Integer ']'
RepeatedEvent ::= InstantEvent ( '{' Integer ',' Integer '}' | '*' )
```

We formally define an event variable as $e_i(t)$ where at an instant t the event occurs.

Event Sequence

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

$$(e_1, e_2, ..., e_n) \equiv \exists t_1 < t_2 < ... < t_n, e_1(t_1) \wedge e_2(t_2) \wedge .... \wedge e_n(t_n) \tag{2.1}$$

where e_i is an event label and t_i is arrival times of SIP messages.

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 latest SIP message is dropped.

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

Note that we can use the short cut-notation to describe event sequence. We can use integers or the "*" to denote a repetition of an event as shown in the following example. In the first rule event *ev_invite* has to be repeated 0 or more times. In the second rule, the event has to be repeated exactly 2 times. In the third rule, the event has to be repeated 2 to 3 times. In the fourth rule,the event has to be repeated at least 3 times. In the last tuple, the event has to be repeated up to 3 times.

```
1 (ev_invite{*},ev_200_OK,ev_ack)
2 (ev_invite{2},ev_200_OK,ev_ack)
3 (ev_invite{2,3},ev_200_OK,ev_ack)
4 (ev_invite{3,},ev_200_OK,ev_ack)
5 (ev_invite{,3},ev_200_OK,ev_ack)
```

Embedded events

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

$$(e_1(e_2, e_3, ..., e_n)) \equiv \exists t_i, e_1(t_i) \wedge e_2(t_i) \wedge ... \wedge e_n(t_i) \tag{2.2}$$

For example a malformed INVITE message is defined using two embedded events. The first event is the arrival of an INVITE message and the second is generated by a malformed field within the message. Rule 2.3.5 depicts the usage of such patterns to prevent a malformed INVITE message to be passed to the target. In this example the events *ev_Invite* and *malformed* happen at the same time.

Rule 4 An illustration example of the usage of the embedded event pattern.

```
DEFINITION EmbeddedPatternDefs BEGIN
WHEN sip:request.method @MATCH "^INVITE$" -> LET:EVENT ev_Invite;
WHEN SIP:body.connection !@MATCH
     "^IN(\s+)IP4(\s+)(\d+)\.(\d+)\.(\d+)\.(\d+)" -> {
        LET:EVENT malformed;
}
DEFINITION END

VETO EmbeddedPattern USES EmbeddedPatternDefs BEGIN
(ev_Invite(malformed)) -> DROP;
VETO END
```

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:

$$\begin{array}{l}(e_1, .., e_{j-1},\ e_j, e_{j+1}, ..., e_n) \equiv \\ \exists t_1 < .. < t_{j-1} < t_{j+1} < ... < t_n, e_1(t_1) \wedge ... \wedge (e_n(t_n)) \\ \wedge(\forall t_1 <= t_i <= t_n, \neg e_j(t_i))\end{array} \tag{2.3}$$

When the sequence is empty, the negation pattern specifies any event different from the specified event. Rule 5 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 en 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 up the contacts list with the values of the field contact of a SIP message.

Temporal patterns

VeTo provides the timewindow 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, Rule 6 the VeTo block prevents from a flooding attack where 1000 INVITE messages arrive within less than one second. We use temporal event patterns to describe a broken handshaking vulnerability as defined in [16]. In this attack a

Rule 5 An illustration example of the usage of a negation event pattern.

```
DEFINITION NegationPatternDefs BEGIN
LET: SET[sip:headers.contact] contacts;
WHEN sip:request.method @MATCH "^BYE$" -> LET:EVENT Ev_BYE;
DEFINITION END

VETO NegationPattern USES NegationPatternDefs BEGIN
(~Ev_BYE ) -> STORE:contacts;
VETO END
```

Rule 6 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_Event_Pattern USES SIPMessages BEGIN
([ev_invite[*1000],1]) -> DROP;
VETO END
```

caller sends an INVITE and when receiving the 200 OK, it doesn't acknowledge it with an ACK. The rules depicted in Rule 7 describe the prevention against this vulnerability exploitation.

Rule 7 The VeTo protection block against the handshake attack.

```
1  DEFINITION HandShackingDefs BEGIN
2  LET: SET[sip:headers.from] black_list;
3  WHEN SIP:response.code @MATCH "^200\s+OK" -> LET:EVENT ev_200_OK;
4  WHEN SIP:response.method @MATCH "^INVITE$" -> LET:EVENT ev_InviteMethod;
5  WHEN SIP:request.method @MATCH "^ACK$" -> LET:EVENT ev_ack;
6  DEFINITION END
7
8  VETO HAND_SHACKING_Vulnerability uses HandShackingDefs BEGIN
9  (ev_200_OK(ev_InviteMethod),~[ev_ack,1]) -> {
10      STORE:black_list;
11      DROP;
12 }
13 (*) -> IF (black_list @CONTAINS "sip:headers.from") {
14     DROP;
15 }
16 VETO END
```

The first event rule in the veto block *HAND_SHAKING_Vulnerability* feeds the collection variable *black_list* with the URI of the sources of the uncompleted handshake messages. This list is used by the second rule shown in line 12 from the same VeTo block to disallow the attacker from sending more broken handshakes.

2.3.6 More event patterns examples

In the following table, we summarize the use of event patterns.

Event pattern	Meaning
$(e1)$	e1 happens
$(e1, e2)$	e2 happens immediately after e1
$\sim e1$	any event but e1 happens
$e1(e2)$	e2 and e2 happen at the same time
$\sim e1(e2)$ or $e2(\sim e1)$	e2 happens while e1 does not
$e1*, e1\{0,\}$	e1 is repeated 0 or more times
$(e1, [e2, 2])$	e2 should happen within 2 seconds after e1
$([e1\{2\}, 1])$	e1 should happen twice within one second
$(e1, \sim [e2, 2])$	e2 should not happen within 2 seconds after e1

2.3.7 State variables

A state variable is used to keep a value over many cycle executions of VeTo rules within a dialog. It either takes user-defined values to express a property of the behavior of the SIP protocol or it takes its values from the runtime to express a SIP entity state. A state type may be generic or specific to a SIP entity. A specific state type is related to a SIP element name. The SIP element is either the type of the user agent (uac or uas) or a resource name defined in the context block referenced by a protection block. In this case, the state variable tracks the state of the SIP element according to the standard SIP state machines as presented in [20]. A state variable is defined using the following syntax:

```
'let' ':' [UA '.' ] 'state' StateVariable
UA::= 'uas' | 'uac' | ResourceName
```

A state variable has *INIT* as an initial value after it has been created. During cycle execution of VeTo rules, a generic state variable values are altered using the *assign* action on the right part of an event rule. We note, that between different cycles execution, the state variable keeps its previous value until its is altered by an *assign* action of an executed rule. For example, Rule 8 describes an improper message INVITE in a UAS confirmed state [20].

Rule 8 A VeTo protection block against an out of state INVITE message.

```
DEFINITION OutStateDefs BEGIN
LET: uas.state UAS_State;
WHEN sip:request.method @MATCH "^INVITE$ -> LET: EVENT ev_Invite;
DEFINITION END

VETO Out_Of_State@{*} USES outStateDefs BEGIN
(ev_Invite) -> IF (UAS_State @EQ "Confirmed") {
DROP;
}
VETO END
```

We firstly define a variable named *UAS_State* of type uas.state which tracks the states of UAS entity involved in a SIP dialog. The instance of the variable *UAS_State* is created an initialized with the *INIT*

value When a dialog is observed the runtime starts to assign the SIP protocol states of the UAS into the state variable *UAS_State*.

A state variable may also be generic where it is managed by the user. Therefore, it takes user defined values. In its definition syntax we omitted the *UA* part.

2.4 Variables scope and extent

Each *VeTo* 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 any event block that references its definition block.

The extent of a VeTo variable is either message or dialog. Typically, the predefined variables named as *SIP:request.**, *SIP:response.** and *SIP:headers.** have an extent over a message. They take a new value every time when a new SIP message is observed. The variables of type event have also a message extent. The variables declared within a VeTo block have a extent over a dialog. Their values are associated within each existing dialog. A variable that takes its values over many dialogs is declared using the keyword *global* followed by its type and name. Rule 9 depicts an example of using a global variable named *BlockList*. The *BlockList* variable is defined using the keyword global. It means that only one instance of the variable is created over existing dialogs. However, the variable *PeersList* has as many instances as existing dialogs. The event pattern *ev_Malicious* drops every SIP message with a contact field value that belongs to the global set *BlockList*.

Rule 9 Prevention from a SIP BYE attack and its subsequent messages.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "BYE" -> LET: EVENT Ev_BYE;
DEFINITION END

DEFINITION BYEAttackDefs BEGIN
LET: GLOBAL SET[sip:message.contact] BlockList;
LET: SET[sip:message.contact] PeersList;
DEFINITION END

VETO BYE_Attack USES BYEAttackDefs, SIPMessages BEGIN
(~Ev_BYE) -> STORE:PeersList;
(Ev_BYE)  ->
IF (PeersList !@CONTAINS sip:message.contact){
STORE:BlockList;
DROP;
}
(*) -> IF (BlockList @CONTAINS sip:message.contact) {
DROP;
       }
VETO END
```

2.5 VeTo Conditions

The left part of veto rules relies on conditions to match patterns or apply logical and mathematical operators. The satisfaction of a condition implies the execution of one or many actions described in the right side of a rule. The purposes of a condition are to detect a pattern within a field value of the current screened SIP message, to check the presence or the absence of a pattern or to check relationship between variables.

2.5.1 Definition of conditions

A condition is defined using the following syntax:

```
Condition::= Assertion * (LogicalOperator Assertion)
LogicalOperator::='||' | '&&'
SubCondition::= Variable Operator Pattern
```

A condition may contain a single or a set of assertions related using conjunction denoted by the symbol && and disjunction denoted by the symbol ||||. It consists of a conditional relation between an variable, an operator and a pattern. The object represents the input of the condition on which we apply an operator. The operator takes as an argument a pattern. We can use any defined stateless or stateful object but with respect to the semantics of the used operator. The output of a condition is a boolean value (True, False).

2.5.2 Semantics of conditions

A Condition relies on different types of operators between a variable and a pattern:

- The matching operator denoted by the symbol "*@MATCH*" is used to match a SIP message field value against a regular expression based pattern.
- The collection operators are used against collections to verify the presence or the absence of an element. The *"@CONTAINS"* operator verifies if a collection variable contains a specific value. The "@IN" operator verifies if a specific value is present in a particular collection. The two operators are equivalent.
- The comparison and equality operators are used to compare an object against a value. These operators are : *@EQ* to denote equality and *@GE* to denote greater or equal than. We note that we can use the negation symbol "*!*" to obtain other relations between variables and values.

2.5.3 Examples of Conditions

Below examples to illustrate the use of conditions within VeTo rules are given.

Simple rule conditions The first rule ensure that when we have observed an INVITE message, an *ev_Invite* event has been created. The second rule checks the value of the protocol version field of the message body. If the value of this field is empty, an event named *ev_Malformed* is created.

```
1 DEFINITION TwinkleDefintion BEGIN
2 WHEN sip:request.method @MATCH "^INVITE" -> LET:EVENT ev_Invite;
3 WHEN sip:body.protocol_version @MATCH "^$" -> LET:EVENT ev_Malformed;
4 DEFINITION END
```

Event patterns conditions The prevention event pattern using the definition described above is depicted in Rule 10. The pattern specifies that we observed an *Ev_Invite* event that embeds an *Ev_Malformed* event and they are followed by any event. The INVITE message and all its subsequent messages within the dialog are dropped.

Rule 10 A prevention block from a malformed INVITE message exploitation.

```
VETO Twinkle_No_Argument_SDP USES TwinkleDefintion BEGIN
(Ev_Invite(Ev_Malformed),*) -> DROP;
VETO END
```

2.6 VeTo actions

The Veto language provides two types of actions: disruptive and non disruptive actions. The disruptive action acts on the SIP message by dropping it. The non disruptive action mainly creates new variables or changes the value of a variable. An action drops a SIP message by using the *drop* statement. An action creates new variables by using the statement *Let*. A VeTo rule may contain many actions. They are grouped between brackets.

2.6.1 *LET* action

This action defines variables that are created by action parts of a rule. At the compilation stage, this statement creates and initializes the variable. At runtime, when its reference rule is fired, this statement updates the variable with new values according to its type. For example, the *LET* action may create an instance of a variable of type event. After it has been created, the variable is consumed by the event patterns where it is referenced.

2.6.2 *STORE* action

The STORE action has the role to provide persistence to the values of a predefined VeTo variable in a collection. It firstly creates the collection variable that stores the values, and then updates it with the current value of the predefined variable. At runtime, the variable instance is created and the values are updated from the current SIP message of the current SIP dialog.

2.6.3 *ASSIGN* action

To alter the value of a defined variable, the VeTo language provides the *Assign* action. The action takes as arguments the name of the object followed by the assignment operator "=" and the new value. The variable keeps its values over the current SIP dialog messages or over multiple dialogs.

2.6.4 *APPLY* Action

The APPLY action executes a built-in VeTo function to its arguments. It updates and returns the value of the result variable assigned to a VeTo function (counter).

2.6.5 *DROP* action

When a vulnerability is detected, the drop action allows us to drop the current SIP message matching the last pre-condition of the running rule.

2.7 Implementation issues

In this section, we present the implementation algorithms developed to support the VeTo runtime.

2.7.1 Rules compilation and checking

A naive approach to rule compilation and implementation compiles rules as linked lists and checks each rule while looping. This approach performs slowly with an important number of rules to check. Therefore, we propose the use of the Rete algorithm [11] for VeTo rules evaluation.

The *Rete* algorithm uses a data flow network to represent the conditions of rules. The dataflow network is structured in a acyclic graph linking variable names, patterns, relations and rules. Each VeTo rule is a composition of conditions and actions. VeTo conditions are conjunction of predefined variables, a matching opertaor and regular expressions. Relations show possible evaluation of variables. The nodes representing variables are called *alpha-nodes* and nodes representing matching relations are called *beta-nodes*. The Rete algorithm uses a working memory to insert and retract variables. In our work, we consider that the algorithm has many working memories where each of them represents an active SIP dialog. When a dialog is active, its working memory is activated. A working memory contains instances of variables defined by the set of rules to be evaluated. Each variable instance enters the Rete network at a single node and is then propagated through the network until it arrives at a terminal node. The alpha part of the network performs simple conditional tests on a working memory elements to be inserted. Within the alpha part, each branch of alpha nodes terminates at a memory, called an *alpha* memory. These memories store variables instances that match each condition in each node in a given node branch. Each beta node has a left and right input. It sends its output to a beta memory. During one cycle identified by a SIP dialog identifier, the engine will find all possible matching rules for the current message. The set of rules is activated on an agenda. The engine determines an order in which rules are fired. Currently, the order may be based on rules order as written by the rules author. Other criteria may be defined to resolve the conflict between rules. The engine now fires the rules and executes actions associated with rules.

Let us now consider the handling of VeTo rules in a Rete graph. When a SIP message is observed by the runtime, the working memory of the current dialog of the message is activated. The working memory contains instances of variables from the current set of rules. Then, for each existing variable instance, we search an applicable rule. When a rule is found, it is activated to be fired. The alpha network part starts handling the matching of predefined variables instances against regular expressions. If it successfully matches, its value is stored in an alpha memory. To illustrate the usage of the Rete algorithm in VeTo, lets consider the blocks depicted in Rule 11.

The two blocks of event rules prevents a SIP phone from the exploit of two vulnerabilities. The first *SJPhone_Message_Chopped* block prevents from a vulnerability where the content length value of the body of an INVITE message is different from its real content length as observed in the message. The second block *SJPhone_Invalid_IP* checks the conformance of the connection parameter of an INVITE message to its specification. We observe that the two blocks require an INVITE message as a first condition. The two blocks use the block *SJPhoneData* that contains the definition of events involved in the two vulnerabilities.

The set of rules from the definition block is compiled into a discrimination network as depicted in Figure 2.1. At runtime, when a SIP message arrives, the predefined variables from the left side of rules are added to the working memory with their respective values. In the above example, the working memory contains

Rule 11 Prevention from two SIP vulnerabilities targeting the SJphone SIP software.

```
CONTEXT SJPhone BEGIN
TARGET => udp:192.168.1.25:*;
CONTEXT END

DEFINITION SJPhoneData BEGIN
WHEN sip:request.method @MATCH "^INVITE" -> LET:EVENT ev_Invite;
WHEN sip:body.connection !@MATCH "^IN(\s+)IP4(\s+)(\d+)\.(\d+)\.(\d+)\.(\d+)" ->
  LET:EVENT malformed;
DEFINITION END

VETO SJPhone_Message_Chopped@SJPhone USES SJPhoneData BEGIN
(ev_Invite) -> IF (SIP:headers.Content_Length !@EQ "SIP:body.length")
                     DROP;
VETO END

VETO SJPhone_Invalid_IP@SJPhone USES SJPhoneData BEGIN
(ev_Invite(Malformed)) -> DROP;
VETO END
```

the method field, the content length and the connection parameter. If the predefined variables match their respective patterns and satisfy a rule, new variables are added to the working memory. For example, if the method field matches the *INVITE* pattern, the first network branch is fired and an instance of the event *ev_INVITE* is added to the working memory.

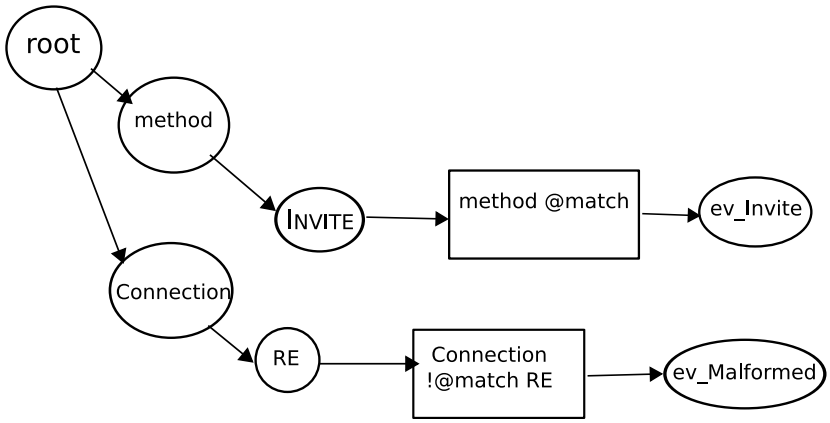

Figure 2.1: The Rete network for the prevention rules targeting the SJphone SIP stack

We observe that the two first rules are merged within a single branch in the Rete network.

2.7.2 Events pattern matching

The event patterns in VeTo provide logical and temporal correlations between events to prevent the exploit of a vulnerability. When an event pattern is detected, the action part of the rule is executed. The event pattern may be detected from one or many SIP messages within a SIP dialog. However, the rules with a simple matching left side using predefined variables are checked from a single SIP message. Therefore, we have to integrate both the matching of event patterns and simple patterns in a single network. The events in VeTo are created using the action *let*. Then, they are consumed to be matched against a pattern defined in the left side of a rule. A naive approach might be to test events against the matched patterns at each execution cycle. This approach is inefficient when dealing with a large number of event patterns. A more interesting approach relies on graphs to event patterns matching. Lets consider, the following example to illustrate our approach for event patterns matching.

```
1  DEFINITION SIPMessages BEGIN
2  when sip:request.method @MATCH "INVITE" -> LET: EVENT ev1;
3  WHEN sip:response.method @MATCH "200" -> LET: EVENT ev2;
4  WHEN sip:request.method @MATCH "ACK" -> let:event ev3;
5  WHEN sip:content_length @EQ "0" -> LET: EVENT ev4;
6  DEFINITION END
7
8  VETO EventsExample@{*} USES SIPMessages BEGIN
9  (ev1,ev2) -> DROP;
10 (ev1(ev4)) -> DROP;
11 (ev2,[ev3,1]) -> DROP;
12 VETO END
```

The above example uses three event patterns defined by three rules of the events block *EventsExample*. The event patterns rely on four events ev1, ev2, ev3 and ev4. These events are defined by rules 1, 2, 3 and 4 of the definition block *SIPMessages*. These rules use the matching operator over a SIP message fields and different regular expressions. We observe that each event pattern is embedded within and Event-Condition-Action rule. The ECA rules were well used for expressing rules in active database management systems. Many approaches [14] exist in the context of active DBMS to filter events within an ECA rule. We distinguish automata, Perti nets and graphs.

In [1] authors use event graph to detect event composition. According to them, their approach is more efficient than other approaches like Petri Nets or extended automata. We use a similar approach based on graphs to detect event patterns expressed by VeTo rules. Therefore, we use an event tree to detect each pattern and an event detection graph to merge and detect the overall patterns. Each leaf node in an event tree is a simple event like *ev1* and internal nodes represent event patterns. The simple events coming from the Rete Network are fed into the graph at the bottom. According to the connected nodes, the events are either discarded or propagated. By using event graphs, the event patterns referenced by VeTo rules are optimized since event nodes are shared by many patterns.

Figure 2.2 represents the event patterns used by the rules of the above example. The bottom of the graph contains the set of events. The middle nodes and the top nodes of the graph contains the event operators such as sequence, embedded and time window to be applied on the triggered events. The event patterns are detected according to this graph. when an event occurs, which means that a *let* action is activated on the event name, and it's current dialog instance propagated through the graph. The paired events in each

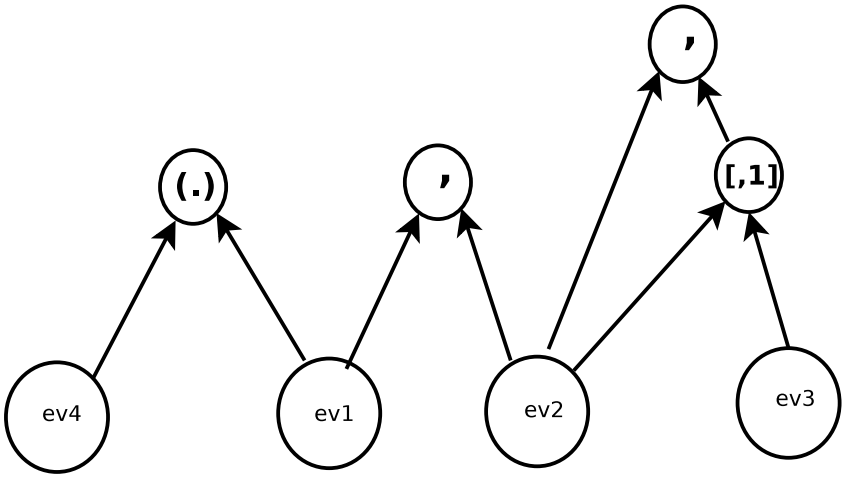

Figure 2.2: The detection graph of event patterns shown in the above illustration example.

operator node are propagated over the graph until a pattern is satisfied. When the pattern is satisfied, the respective rule is fired and the set of actions is executed.

Chapter 3

VeTo Testing

In this part, we present a more detailed complete examples of the Veto language to express severeal preventions from known vulnerabilities of the SIP protocol. The examples are drawn from discovered vulnerabilities by a fuzzing process [4] and others were published in bulletin alerts [25].

3.1 Malformed messages vulnerabilities

SIP malformed messages vulnerabilities are of type stateless and did not involve multiple SIP messages over a dialog. This type of vulnerabilities is quite easy to write in the VeTo language. An incorrect grammar SIP message, oversized field value, an invalid message or field name are expressed by regular expressions and the matching operator. Following, we describe some examples of malformed SIP packets as described in [5]

3.1.1 Incorrect Grammar Message

The RFC 3261 [20] defines the grammatical structure of a SIP message. The detection of incorrect syntax of fields may be derived directly from the SIP ABNF. For example the Call ID field is defined as :

```
Call-ID=("Call-ID"/"i") HCOLON callid
callid = word ["@" word]
```

This may be easily translated into a simple regular expression in the Veto rule condition part with the matching operator "*@match*". Therefore each time the event is satisfied, the SIP message is dropped.

```
DEFINITION MalformedDefs BEGIN
WHEN sip:headers.call-ID !@MATCH "^\s*(Call-ID|i)\s*:\s*w+[@w+]$" -> LET: EVENT malformedCallID;
DEFINITION END

VETO VulnerbaleCallID USES MalformedDefs BEGIN
(malformedCallID) -> DROP;
VETO END
```

Here, we only need to write rules for the correct grammar of SIP messages and any message does not comply with this grammar is dropped. Other rules to detect oversized field values or a missing mandatory field could be easily written using veto rules and regular expressions.

For example, it was reported by a fuzzing testing that an empty method field of a SIP request may takes down a SIP stack implementation [4]. This type of vulnerability is mainly due to an implementation flaw where the software has not been implemented robustly. The rule below detects an empty method field and drops the SIP message that triggers the event.

```
DEFINITION MalformedDefs BEGIN
WHEN sip:request.method" @MATCH "^$" -> LET: EVENT malformedMethod;
DEFINITION END

VETO VulnerbaleMethod USES MalformedDefs BEGIN
(malformedMethod) -> DROP;
VETO END
```

3.1.2 Cross-site scripting attacks over SIP

Cross-site scripting (XSS) is a type of vulnerability, usually found in web applications which allow code injection by malicious web users into the web pages viewed by other users. Recently, it has been proved that an attacker may uses the SIP to start an XSS attack and owns an internal network access. A quick answer could be: use a buffer overflows and do it. While buffer overflows in SIP stacks do exists, most of them are difficult to exploit because they are affecting embedded devices with custom architectures and operating systems. However, most VoIP devices have embedded web servers that are typically used to configure them, or to allow the user to see the missed calls, and all the call log history. The important issue is that, the user will check the missed calls and other device related information from her machine, which is usually on the internal network. In this post, I will describe how XSS injection can be done with SIP and a vulnerable integrated web server. The Linksys SPA-941 (Version 5.1.8) phone has an integrated web server that allows for configuration and call history checking. An XSS vulnerability allows a malicious entity to perform XSS injection because the "FROM" field coming from the SIP message is not properly filtered. By sending a crafted SIP packet with the FROM field set to : *"¡script x=''' ¡sip:'src='http://baloo/beef/y.js'¿192.168.1.9:5060¿;tag=1",*

the beef attack tool can be launched against the internal machine of the user. Obviously any other XSS related attack (XSS proxy, Beef, Attack API, Jikto) can be similarly performed.

Rule 12 Prevention from a SQL injection code within a from field of a SIP message.

```
DEFINITION MalformedDefs BEGIN
WHEN sip:headers.from @MATCH "script" -> LET: EVENT malformedFrom;
DEFINITION END

VETO SQL_Injection UESES MalformedDefs BEGIN
(malformedFrom) -> DROP;
VETO END
```

The bloc of protection depicted in 12 provides a rule which verifies the presence of the keyword *script* within a *from* field of a SIP message.

The different veto blocks described above to prevent from the exploit of a malformed SIP message may be combined in a single block. Rule in 13 depicts the definition block and the event block of such vulnerability. We observe that in the definition block, the event *MalformedMsg* is created from different matching rules. It is equivalent to a disjunction operator between these rules. However, the event block *MalformedMessage* contains a single rule.

Rule 13 Prevention from a malformed SIP message.

```
DEFINITION MalformedDefs BEGIN
WHEN sip:headers.from @MATCH "script" -> LET: EVENT MalformedMsg;
WHEN sip:request.method" @MATCH "^$" -> LET: EVENT MalformedMsg;
WHEN sip:headers.callid !@MATCH
 "^\s*(Call-ID|i)\s*:\s*w+[@w+]$" ->
 LET: EVENT MalformedMsg;
DEFINITION END

VETO MalformedMessage USES MalformedDefs BEGIN
(MalformedMsg) -> DROP;
VETO END
```

3.2 Invalid semantics

In this category of vulnerabilities, the SIP message is well formed, but it reveals to an abnormal semantic. For example, the use of the loopback address (127.0.0.1) in a sip message, or an out of state received SIP message [6]. Rule 14 prevents from an improper INVITE message received by a callee in the confirmed state of the SIP server state machine.

Rule 14 Prevention from an improper INVITE message in a UAS confirmed state.

```
DEFINITION SIPMessages BEGIN
LET:uas.state Callee_State;
WHEN sip:request.method @MATCH "^INVITE" -> LET: EVENT ev_Invite;
DEFINITION END

VETO Improper_INVITE USES SIPMessages BEGIN
(ev_Invite) -> IF (Callee_State @EQ "confirmed") {
DROP;
}
VETO END
```

The definition block creates a state variable named *Calle_State* which tracks the state value of an UAS entity within a dialog. The second rule from the same block, creates and event *ev_Invite* when an INVITE SIP message is observed. The veto block *Imporper_INVITE* drop the SIP message, if the *ev_Invite* event occurs and the *callee_state* is equal to *confirmed*.

3.2.1 Out of state message

An out of state message is an unexpected message in the SIP protocol state machines. For example, we may observe a 200 OK response message without an associated request. The vulnerability is described in Rule 15. The rules in the definition block *SIPMessages* creates an event named *ev_Request* when an INVITE, a BYE or a Register messages are observed. The event rule within the *Out_of_State* block checks the arrival of the *ev_200* without any *ev_request* event. In such case, the SIP message is dropped.

Rule 15 Prevention from a 200 OK out of state response message.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "INVITE" -> LET: EVENT ev_Request;
WHEN sip:request.method @MATCH "BYE"    -> LET: EVENT ev_Request;
WHEN sip:request.method @MATCH "REGISTER"    -> LET: EVENT ev_Request;
WHEN sip:response.code @MATCH -> "^200" -> LET: EVENT ev_200;
DEFINITION END

VETO Out_of_State UESES SIPMessages BEGIN
(~ev_request,*,ev_200)" -> DROP;
VETO END
```

3.3 Flooding attacks

In [6], the author was interested to SIP-specific DoS attacks that involve flooding SIP entities with illegitimate SIP messages. He proposes a detection method that relies on thresholds. The author specifies four threshold parameters to detect flooding attacks. The most trivial one is a threshold on the number of INVITE message allowed per dialog to a particular URI. Rule 16 describes such vulnerability. Firstly, we track the target URI of each INVITE message within the collection *targets*. We count the number of INVITE message to each target from the collection using the function *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.

The most interesting is the upper bound of the number of allowed transactions per node. Firstly, We need to define a collection that tracks transactions per node. We also need a counter to track their number. Rule 17 describes such vulnerability.

Another threshold type, described by the author, is the upper bound on the number of allowed messages per transaction. In this case we use a collection variable to record transactions. We associate to each transaction a counter to count the number of request and response messages. When this number exceeds a particular threshold, the subsequence messages are dropped. Rule 18 depicts the prevention from such attack. The event pattern shown in line 10 describes the occurrence of an *event_Request* followed by zero or many *ev_Response* responses using the kleen closure *[*]*.

3.4 Implementation flaws specific attacks

Attack may occurs on a specific implementation on a particular SIP devices to cause unexpected behavior, usually the device takes down. In [4], authors have identified many of these flaws on a wide range of available open source and commercial SIP implementations. Some SIP phones implementation are sensitive to a particular sequence of SIP messages. Such sequence may takes down the SIP phone implementation. Rule 19 prevents a particular SIP phone, from executing a such vulnerable sequence of SIP messages.

Rule 16 Prevention from an INVITE flooding attack against a particular target URI.

```
DEFINITION FloodingDefs BEGIN
LET: SET[sip:headers.uri.addr] targets;
LET: COUNTER(10,60000) targets.count;
DEFINITION END

VETO Flooding_By_Target USES SIPMessages, FloodingDefs BEGIN
(ev_Invite) -> {
        STORE:targets;
        APPLY:targets.count;
        IF (targets.count @GE 100) {
            DROP;
        }

}
VETO END
```

Rule 17 Prevention from a transaction based flooding attack.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method !@MATCH "^$" -> LET: EVENT ev_Request;
DEFINITION END

DEFINITION FloodingDefs BEGIN
LET: SET[sip:headers.branch] transactions;
LET: COUNTER(1,60000) transactions.count;
DEFINITION END

VETO Flooding_By_Transaction UESES FloodingDefs, SIPMessages BEGIN
(ev_Request) -> {
         STORE:transactions;
         APPLY:transactions.count;
         IF (transaction.count @GE "10") {
            DROP;
         }
}
VETO END
```

3.5 Legitimate SIP messages based vulnerabilities

Legitimate SIP vulnerabilities are a particular type of vulnerabilities that rely on legitimate SIP messages and semantics. However, the vulnerability occurs when a combination of SIP messages is exchanged between SIP entities. In such case a SIP feature is tricked to arise an abnormal or unexpected behavior on a targeted SIP device.

Rule 18 Prevention from a per message transaction based flooding attack.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method !@MATCH "^$" -> LET:EVENT ev_Request;
WHEN sip:response.code !@MATCH "^$" -> LET: EVENT ev_Response;
DEFINITION END

DEFINITION FloodingDefs BEGIN
LET: SET[sip:headers.branch] transactions;
LET: COUNTER(1,60000) transactionsMsg.count;
DEFINITION END


VETO Flooding_Transactions_Messages USES SIPMessages, FloodingDefs BEGIN
(ev_Request,ev_Response[*]) -> {
  TORE:transactions;
        APPLY:transactionsMsg.count;
        IF (transactionsMsg.count @GE "10") {
            DROP;
        }

}
```

Rule 19 Prevention from a vulnerable sequence of events targeting a Cisco SIP phone.

```
CONTEXT CISCO_7940 BEGIN
TARGET => udp:192.168.1:*;
CONTEXT END

DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "^INVITE$" -> LET:EVENT ev_Invite;
WHEN sip:request.method @MATCH "^OPTIONS" -> LET: EVENT ev_Options;
WHEN sip:response.code @MATCH "^200" -> LET: EVENT ev_200;
WHEN sip:response.code @MATCH ""^481" -> LET:EVENT ev_481;
DEFINITION END

VETO Cisco_flaw@{CISCO_7940} USES SIPMessages BEGIN
(ev_Invite,ev_481,ev_Options,ev_200,ev_481,ev_Options) -> DROP;
VETO END
```

3.5.1 SIP Method based attacks

This kind of attack uses a legitimate SIP request to terminate or modify an existing dialog. In the *CANCEL* attack, a crafted *CANCEL* request is sent before the final response of a dialog/transaction, thereby terminating the dialog prematurely, causing a DoS. The *Cancel* attack involves the generation of a legitimate *CANCEL* request to terminate an on-going dialog between two SIP peers. Rule 20 presents the set of rules to prevent from such attack.

Rule 20 Prevention from a vulnerable CANCEL message.

```
DEFINITION SIPMessages BEGIN
WHEN  SIP:request.method @MATCH "^CANCEL" -> LET:EVENT ev_CANCEL;
DEFINITION END

DEFINITION VulnerableCancelDefs BEGIN
LET: SET[sip:headers.contact] Contact_List;
DEFINITION END

VETO VulnerbaleCANCEL USES SIPMessages, VulnerableCancelDefs BEGIN
(~ev_CANCEL) -> STORE:contact_list;
(ev_CANCEL) -> IF (sip:headers.contact !@in contacts_List) {
                DROP;
}
VETO END
```

3.5.2 The SIP DNS flooding attack

In [27], authors describe a SIP attack where a server is flooded with requests addressed at irresolvable domain names. The attack targets SIP proxies where header field used for routing contains an irresolvable URI. The attacker uses a URI, of which it is sure that its mapping will not be in the cache of a name server. Rule 21 depicts the set of rules that prevents from such attack. Firstly, we define a collection called *domains* to store the requested domains from a register SIP messages. We associate a counter to count the number of register messages to a particular domain. When the domain is resolved we assign the value 0 to its associated counter. If the number of attempts to resolve a particular domains exceeds the value 2 we drop all subsequent SIP messages targeting such SIP domain.

Rule 21 Prevention from a SIP DNS flooding attack.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "^REGISTER" -> LET: EVENT ev_Register;
WHEN sip:response.code @MATCH "^200" -> LET: EVENT ev_Response;
DEFINITION END

DEFINITION DNSFloodingDefs BEGIN
LET: SET[sip:headers.uri.domain] domains;
LET: COUNTER() domains.score;
DEFINITION END

VETO SIP_Proxy_DNS_Flooding USES SIPMessages, DNSFloodingDefs BEGIN
(ev_Register) -> {
      STORE:domains;
      APPLY:domains.score;
      IF (domains.score @GE "2") -> DROP;
}
(Ev_Register,Ev_Response) -> ASSIGN:domains.score=0;
VETO END
```

3.5.3 Ringing-based DoS attack

In [7], the authors describe the ringing attack where an attacker exploit the semantics of the SIP protocol by intentionally increasing the amount of transaction state that must be maintained by a stateful proxy. A stateful proxy with limited resources can only maintain a finite amount of transaction state before service disruption start to occur. If one or more UACs repeatedly send INVITE requests to one or more excessively ringing UASs, then the stateful proxy can potentially become the target of what we call Ringing-based DoS attack. The authors present a solution based on an algorithm to be added to a SIP proxy implementation. As explained above, patching is an efficient way to stop such flaw, but it is known that people have been reluctant to patch their systems immediately, because patches are perceived to be unreliable and disruptive to apply [26]. Therefore, using the Veto language and SecSip is useful to address this attack before patches are applied. The following rules written in the Veto language expresses the solution proposed by the authors in [7]. The idea is to drop transactions suspected of involving excessively ringing UASs when the proxy is under heavy load. If the number of concurrent transaction is greater than T2, then all transaction between T2 up to the oldest transaction which are older than MRTT, will be canceled. The set of rules that prevents from such attack is depicted in Rule 22.

Rule 22 Prevention from ringing-based DoS attack.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "^INVITE" -> LET: EVENT ev_Invite;
WHEN sip:response.code @MATCH "^200$" -> LET: EVENT ev_200;
DEFINITION END

DEFNITION RingingDefs BEGIN
LET: SET[sip:request.branch] Transactions;
LET: TIMER() Transactions.timestamp;
LET: COUNTER() Transactions.count;
LET: STATE Transactions.state;
DEFINITION END

VETO Ringing_Attack BEGIN

(ev_Invite) -> {
     STORE:Transactions;
     APPLY:Transactions.timestamp;
     APPLY:Transactions.count;
     IF (transactions.count @GE "T2" &&
         transactions.timestamp @GE "MRTT" &&
         transactions.state !@EQ "Terminated")
         DROP;
}

(ev_200) -> {
      ASSIGN:transactions.state=Terminated;
}

VETO END
```

3.5.4 The SIP Identity baiting attack

This attack [15] relies on a weakness of the SIP identity mechanism described in RFC 4474 [17]. The general form of the attack yields to re-use a harvested signed request from a legitimate party to call a victim. The victim domain receives the request, and verifies the identity signature. The attacker has not changed anything which was signed, so the validation succeeds. The To URI contains the address of the attacker, but the victim domain does not verify that the To domain is valid, nor could it because the request may have simply been forwarded through re-targeting, which is legitimate.

What makes the attack successful is that requests in SIP are routed based on the Request-URI and/or route headers, not the To-URI. One possible solution is to verify that the URI value of the To field matches a trusted list of address of record. Therefore, the set of rules depicted in Rule 23 prevents from such attack. The rule drops all INVITE messages where the To field value does not belongs to the trusted list *AoR*. The AoR list is defined in the context block *SIP_PHONES*.

Rule 23 Prevention from a baiting attack.

```
CONTEXT SIP_PHONES BEGIN
TARGET=> *:192.169.1.*:*;
RESOURCE => AoR={sip:bob@example.com,alice@example.com,...};
CONTEXT END

DEFINITION BaitingDefs BEGIN
WHEN sip:request.uri !@MATCH sip:request.to.uri -> LET: EVENT ev_suspicious;
DEFINITION END

VETO Baiting_Attack@{SIP_PHONES} USES SIPMessages, BaitingDefs BEGIN

(ev_Invite(ev_Suspicious)) -> IF (Aor !@contains SIP:request.to.uri) {
       DROP;
}
VETO END
```

3.5.5 The Misuse-ringing attack

In [9], it was reported a VoIP attack against German home users of the Heise VoIP provider. The users reported that their VoIP phones started ringing in the middle of the night. By examining the resulted SIP Invite message of this attack, it seems to used a branch parameter value that not contains the magic cookie *z9hG4bK*. In addition, the Request URI, To and From domains part contains an invalid IP address *1.2.3.4* and the Call-ID contains spaces. It is easy to mitigate such attack by writing rules that detects such malformed fields and consequently drop respective messages. This solution is inefficient since the vulnerability exploited by the attacker still exists and it is easy for an attack to evade from the malformed fields detection rules. The attack may has different targets either SIP phones or SIP servers. Therefore, we need to contextualize the protection rules according to the target device.

SIP Phones: To receive incoming calls a SIP phones needs to register its address-of-record to a specific contact address to its SIP proxies that serve its domain. Therefore, each incoming call where the request URI does not match the Contact-URI in the register request has to be dropped. The following rules record the Contact-URI within the REGISTER requests. If an invite request contains a request URI that does

not match a registered contact-URI. Thus, the request should be dropped. The veto block *Misuse_Attack*

Rule 24 Prevention from the misuse ringing attack targeting one or many SIP phones.

```
CONTEXT SIP_PHONES BEGIN
TARGET => *:192.169.1.*:*;
RESOURCE => My_SIP_Proxy_Addr=192.168.2.25;
CONTEXT END

DEFINITION MisuseDefs BEGIN
LET: SET[sip:request.contact] contacts;
WHEN sip:message.via.addr !@MATCH My_SIP_Proxy_Addr -> LET: EVENT ev_Unknown;
DEFINITION END

VETO Misuse_Attack@{SIP_PHONES} USES SIPMessages, MisuseDefs BEGIN
(ev_Unknown)  -> DROP;
(ev_register) -> STORE:contacts;
(ev_Invite)   -> IF (contacts !@contains sip:request.uri) {
          DROP;
}
VETO END
```

has a context identified by the name *SIP_HONES*. This context has a target a list of addresses of available SIP phones in the SIP network. Firstly, we define a collection to store the set of registered contacts. The first event rule in line 10 allows a SIP phone to only accept incoming calls from a predefined SIP proxy *My_SIP_Proxy_Addr*. The *My_SIP_Proxy_Addr* is defined in the context block. The last event rule in line 12 drops all INVITE messages to a SIP phone with a request uri that does not belong to the stored list of contact uris.

3.5.6 The re-INVITE message syndrome

The SIP protocol establishes a dialog between two user agents using the offer-answer model. It also provides means to modify a session parameters such as addresses or ports, adding a media stream, and so on. This is accomplished by sending a new INVITE request within the same dialog. This type of INVITE is known as a re-INVITE. The Table 3.1 depicts the major different parameters between an INVITE and a re-INVITE messages. The *To, From, Call-ID, Cseq, and Request-URI* of a re-INVITE are set as same as a regular request. In a re-INVITE message the Cseq value is incremented but it keeps the method name as INVITE. It is considered like after INVITE subsequent requests that contain a Cseq value which is incremented by one of the Cseq of the original request. The request-URI of the re-INVITE message should be equal to the To URI field since it is a mid-dialog request. However, the initial INVITE request request-URI may contains a pre-existing route that the denotes the URI of an outbound proxy for example. The VeTo block in 25 detects a SIP message of type re-INVITE within an existing dialog. A re-INVITE message is basically a new INVITE with a non empty To tag within a dialog.

Authentication abusing attack

In [3], the authors claim that a re-invite message may be used to bypass the authentication mechanisms used in SIP networks. Their described attack relies on the use a triggered re-INVITE message with an authentication that is based on authentication in HTTP [12]. The HTTP authentication abusing attack

Parameters	INVITE	re-INVITE
Dialog scope	new dialog	existing dialog
Initiator	Caller	caller or Callee
To tag	empty	existing To tag
Cseq value	Initial value	incremented
Forking	Yes	Never
UAC state	INIT	Terminated
UAS state	INIT	Terminated
Session description	Initiate	modify
To URI/Request URI	Equal or Different	Equal

Table 3.1: The main different parameters between a regular INVITE and a Re-INVITE messages.

Rule 25 The VeTo block that defines events of INVITE and Re-INVITE messages within an existing dialog.

```
DEFINITION ReInviteDefs BEGIN
WHEN sip:request.method @MATCH "^INVITE$" -> LET: EVENT ev_Invite;
WHEN sip:request.to.tag @MATCH "^$" -> LET: EVENT ev_EmptyTag;
DEFINITION END
```

relies on two weakness. The first is related to the re-INVITE message that might be used as an INVITE request to call a user on another domain. The second weakness is related to the HTTP authentication which do not apply the integrity checking on some sensitive SIP header fields.

The synopsis is as follow: an attacker will issue a call directly to the victim, the victim answers and later on, puts the attacker on hold (transfers him to any other place or uses any other method which requires a re-INVITE). Once the attacker receives the re-INVITE specifying the "On hold", he will immediately request the victim to authenticate. This last authentication may be used by the attacker to impersonate the victim at its own proxy.

The vulnerable event within this attack is to relay an authenticated re-INVITE and use it as an INVITE to initialize a call to a victim.

Herein, we distinguish two cases: in the first case the attacker may replays the SIP re-INVITE message on the same proxy of the victim to initialize a call to another user. This scenario is depicted in Figure 3.1.

In this case, we obtain a replay attack scenario. To address this vulnerability, we provide the set of rules depicted in Rule 26. We need to check each nonce value from the authentication response within the INVITE message against a list of used *nonce* values.

In the second case, the victim and the UA who issues the harvested re-INVITE are not on the same domain. This scenario is depicted in Figure 3.2.

The major vulnerability remains on the routing mechanism of SIP [21]. Either proxy or a UA have no way to validate routing headers that might be inserted by attackers. There is also no framework to associate a call with global route information. As described in the RFC 2617, when the integrity feature is used by the client to be authenticated by the proxy, the authentication digest contains a hash of the entity-body of the re-INVITE message. Therefore, the attacker has to keep the values of the SIP message body unmodified. To place a SIP user "on Hold", the UA issues a re-INVITE message with a new SDP offer that include the attribute *sendonly* on the direction parameter. Thus the used re-INVITE by the attacker to initialize a call to a victim will contain on its SDP offer the attribute *sendonly*.

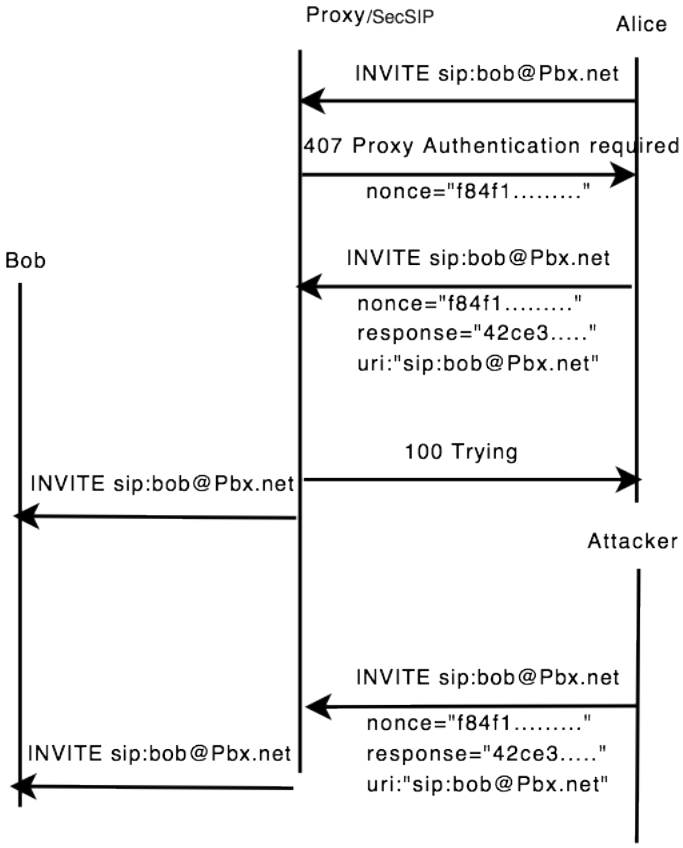

Figure 3.1: The replay attack SIP messages flow.

Rule 26 A prevention from an INVITE replay attack with an older nonce value vulnerability.

```
DEFINITION ReplayDefs BEGIN
LET: GLOBAL SET[sip:headers.proxy-authorization.nonce] Nonce_List;
WHEN sip:response.code @MATCH "^407$" -> LET: EVENT Auth_Required;
DEFINITION END

VETO Replay_Attack@{SIP_PROXY} USES ReInviteDefs,ReInviteDefs BEGIN
(ev_Invite(ev_EmptyTag),Auth_Required) ->  STORE:Nonce_List;

(ev_Invite(ev_EmptyTag)) -> IF (nonce_list @contains SIP:Proxy-Authorization.nonce) {
                              DROP;
}
VETO END
```

The rules to prevent from such vulnerability exploitation due to the re-INVITE syndrome with an HTTP authentication feature are depicted in Rule 27. The protection rules detect an authenticated INVITE message with a sendonly value set on the *sip.body.a* parameter of the SDP offer.

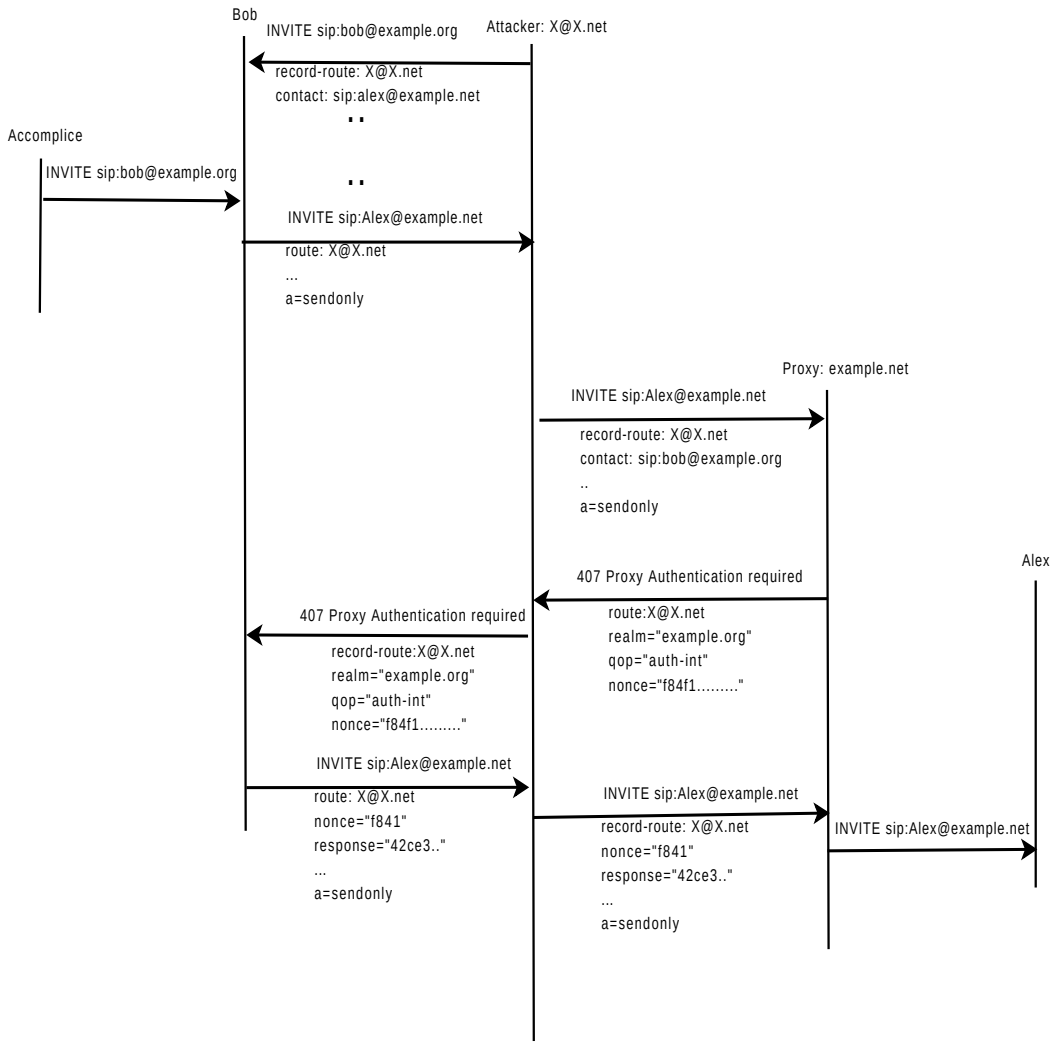

Figure 3.2: The SIP messages flow of the replay attack.

Rule 27 Prevention from a re-INVITE based relay attack.

```
DEFINITION ReplayDefs BEGIN
WHEN sip:body.a @MATCH "^sendonly$" -> LET: EVENT Send_Only;
DEFINITION END

VETO Replay_Attack_Two_Domains USES ReplayDefs,ReInviteDefs BEGIN
(Auth_Required,ev_Invite(ev_EmptyTag,Send_Only)) -> DROP;
VETO END
```

Chapter 4

Conclusion

We have designed a domain specific language, called VeTo to specify protections against the exploitation of existing vulnerabilities in the SIP protocol and/or its implementation. In VeTo a vulnerability preventions is expressed as a set of assertions around the vulnerable point. The vulnerable point is the point where the target system goes wrong. The assertions are the conditions necessary to go wrong. The main question when expressing a vulnerability is *what are the conditions to go towards a vulnerable point ?*. This means describing what relationships hold between various properties to go wrong and violate the intended semantics of the SIP protocol. The specification of these conditional relationships relies on a rule based proactive event based approach.

The resulting language builds on three features which are the definition, the context and the events blocks. These blocks specify respectively the input data of the vulnerability, its surrounding environment and its behavior. Each block is based on declarative rules. The definition block contains pattern matching rules over the current SIP message of the current dialog. The context block defines mainly the target SIP device of the vulnerability. The event block is specific to a particular vulnerability. Its uses rules composed of the tuple events pattern, condition and actions. Each events pattern links different SIP protocol behaviors involved in a specific vulnerability. Our language also provides instructions to record and manipulate protocol histories over messages and dialogs. It also includes temporal support to express temporal properties and relationships among different events involved in a vulnerability specification. We have illustrated the language over several known vulnerabilities published in research works or in alert bulletins. The semantics of the language has been implemented on the SecSIP [2] engine which acts as a proactive point of defense of a SIP infrastructure.

At present VeTo specifications only address 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 technique and software testing and binary analysis have to be used in conjunction to prevent from unknown and zero-day vulnerabilities exploitation.

In future work, we hop to demonstrate formal modeling techniques in order to verify conflicts between VeTo rules an check their consistency, completeness and compactness.

Bibliography

[1] Snoopib: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 59(1):139 – 165, 2006.

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

[3] Humberto Abdelnur, Tigran Avanesov, Michaël Rusinowitch, and Radu State. Abusing SIP Authentication. In *Information Assurance and Security (ISIAS) Information Assurance and Security, 2008. ISIAS '08.*, pages 237–242, Naples Italie, 2008. IEEE.

[4] 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.

[5] Eric Y. Chen and Mitsutaka Itoh. Scalable detection of sip fuzzing attacks. In *SECURWARE '08: Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 114–119, Washington, DC, USA, 2008. IEEE Computer Society.

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

[7] William Conner and Klara Nahrstedt. Protecting sip proxy servers from ringing-based denial-of-service attacks. *Multimedia, International Symposium on*, 0:340–347, 2008.

[8] Ram Dantu and Prakash Kolan. Detecting spam in voip networks. In *SRUTI'05: Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[9] Klaus Darilion. Analysis of a voip attack. http://www.ipcom.at/fileadmin/public/2008-10-22_Analysis_of_a_VoIP_Attack.pdf, october 2008.

[10] Jonathan Davidson, James Peters, Manoj Bhatia, Satish Kalidindi, and Sudipto Mukherjee. *Voice over IP Fundamentals (2nd Edition) (Fundamentals)*. Cisco Press, 2 edition, August 2006.

[11] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. In *Expert systems: a software methodology for modern applications*, pages 324–341, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[12] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[13] Olivier Hersent. *IP Telephony: Deploying VoIP Protocols and IMS Infrastructure, 2nd Edition.* Wiley, October 2010.

[14] Annika Hinze. Efficient filtering of composite events. In *In Proc. of the British National Database Conference*, pages 207–225, 2003.

[15] H. Kaplan and D. Wing. The SIP Identity Baiting Attack. IETF draft, February 2008.

[16] Mohamed Nassar, Saverio Niccolini, Radu State, and Thilo Ewald. Holistic voip intrusion detection and prevention system. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 1–9, New York, NY, USA, 2007. ACM.

[17] J. Peterson and C. Jennings. Enhancements for Authenticated Identity Management in the Session Initiation Protocol (SIP). RFC 4474 (Proposed Standard), August 2006.

[18] Thomas Porter and Jr. Jan Kanclirz. *Practical VoIP Security.* Syngress Publishing, 2006.

[19] Trygve Reenskaug. Models - views - controllers. Technical report, Xerox PARC, December 1979. A scanned version on http://heim.ifi.uio.no/ trygver/mvc/index.html.

[20] 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.

[21] D. Schwartz and J. Barkan. End-to-End Route Management in the Session Initiation Protocol. IETF draft, February 2006.

[22] 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.

[23] 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.

[24] VOIPSA. Voip security and privacy threat taxonomy. http://voipsa.org/Activities/taxonomy.php, October 2005, last checked on november 2011.

[25] VoIPSA.org. VOIPSEC mailing list on VoIP security issues. http://voipsa.org/mailman/listinfo/voipsec_voipsa.org, January 2009, last checked on november 2011.

[26] 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.

[27] Ge Zhang, Sven Ehlert, Thomas Magedanz, and Dorgham Sisalem. Denial of service attack and prevention on sip voip infrastructures using dns flooding. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 57–66, New York, NY, USA, 2007. ACM.

[28] 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.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803