Security testing of SIP implementations

Christian Wieser, Marko Laakso
Department of Electrical and Information Engineering
University of Oulu
Oulu, FIN-90014 University of Oulu
Email: ouspg@ee.oulu.fi

Henning Schulzrinne
Department of Computer Science
Columbia University
1214 Amsterdam Avenue, New York, NY - 10027-7003
Email: hgs@cs.columbia.edu

***Abstract*— The Session Initiation Protocol (SIP) is a signaling protocol for Internet telephony, multimedia conferencing and instant messaging. Although SIP implementations have not yet been widely deployed, the product portfolio is expanding rapidly. We describe a method to assess the robustness of SIP implementation by describing a tool to find vulnerabilities. We prepared the test material and carried out tests against a sample set of existing implementations. Results were reported to the vendors and the test suite was made publicly available. Many of the implementations available for evaluation failed to perform in a robust manner under the test. Some failures had information security implications, and should be considered vulnerabilities.**

I. Introduction

Vulnerabilities in software that endanger confidentiality, accessibility and integrity (information security) of information systems are found on a daily basis [1], [2]. One type of vulnerabilities is introduced during the coding process of the software. Examples of these *implementation level vulnerabilities*, are buffer overflows [3], where a program writes beyond bounds of space allocated for manipulation, and format string vulnerabilities [4], introduced due to the improper handling of variable argument lists.

The Session Initiation Protocol (SIP) is an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls [5]. We have chosen SIP for this work because it has matured from academic interest into a commercially relevant protocol with the potential for wide deployment. However, field usage appears to be in the early stages. Furthermore, SIP is being adopted by the Third Generation Partnership Project (3GPP) as part of the third generation mobile architecture [6]. The SIP family of specifications is expanding and some aspects are under development. The HTTP-like ASCII presentation of the SIP messages may also attract more script-kiddie level hostility than rival protocols - such as H.323. H.323 uses binary (ASN.1) encodings.

The purpose of this paper is to describe a method to assess the robustness of SIP implementations. The test suite will set a baseline to determine vulnerabilities in SIP products and can be used during in-house testing or as part of regression testing. It gives the consumer the possibility to compare products during a product evaluation process or it could be set as a criteria during acceptance testing.

A. Previous work

Different approaches have been taken in testing SIP implementations. A rough categorisation of the tools can be made:

1) Performance testing tools: Performance testing tools, like SIPstone [7], measure the performance of SIP entities or networks.

2) Torture testing tools: SIP interoperability testing events [8] have been held since April 1999. During these biannual events, text messages were collected and released as an (already dated) IETF-draft [9]. It defines 42 valid and invalid messages, describes them and gives directions on how the SIP application should react.

3) Conformance and functional testing tools: A fair amount of work has been invested in the development of conformance and functional test tools, sometimes combining performance testing functionality. A list of available tools can be found at [10]. Also NIST [11] (using XML) and ETSI [12] (using TTCN-3) have released material for conformance and functional testing.

The testing tools identified above focus on conformance and performance, with no or only some robustness testing material. A tool designed for robustness testing would give a baseline for implementation level security, allowing implementors and their customers to find vulnerabilities pro-actively.

The paper is organized as follows: initially we describe the design of the SIP robustness test suite. The results of applying the test suite are given afterwards. Subsequently, we discuss the process of reporting our findings. Finally, a discussion of the findings concludes this paper.

II. Test suite: c07-sip

The framework for creating robustness test suites has been developed in the PROTOS project [13]. One result has been a mini-simulation method [14] and its implementation in a test design and generation tool. This tool have proven to be efficient in testing for vulnerabilities by using a black-box testing method based on syntax testing. In syntax testing, test cases are generated from the input specification and are injected via interfaces [15, p.284].

In contrast to conformance or functional testing, we are not evaluating the reply to an input, ie. we have no test oracle. We ask if the implementation behaves in a secure and robust manner when it is fed with exceptional input and monitor it for failures.

The rest of this section presents the steps to create the test suite and the tests we conducted against available SIP implementations.

A. Standard survey

RFC 3261 specifies the Session Initiation Protocol [5] and serves as protocol specification. Due to our interest in the voice-over-IP usage of SIP, we use RFC 2327 [16] to describe a multimedia session, although no actual multimedia content will be transmitted during the testing.

B. Subject survey

A survey of available implementations was conducted. This includes a diverse selection of implementations in order to gain a better insight into the applications implementing the protocol, and gives a hint of the impact of the potential vulnerabilities.

RFC 3261 identifies several types of SIP entities: the user agent, e.g., a SIP enabled voice-over-IP phone; redirect server, i.e., a user agent server to redirects requests; proxy server, that acts on behalf of other entities; and the registrar, that provides location services.

A subset of the implementations was chosen as a sample set to be tested during the test suite creation and pre-release phases. Reasons for omission of a specific product would be that we could not get an evaluation copy of a product, the licensing contracts were too restrictive, we could not buy it, or we were simply not aware of the product. Altogether we were able to acquire nine implementations, six user agents and three proxy servers.

C. Injection vector survey

In injection vector survey, different methods of delivering the test cases to the implementation are identified and analyzed in Table I.

TABLE I
INJECTION VECTOR SURVEY

Application Protocol	Transport Protocol	Port
SIP	UDP	5060
SIP	TCP	5060
SIP-over-TLS	TCP	5061

All RFC 3261 compliant SIP elements must implement UDP and TCP transports, and in case of UDP the processing of multicast (including broadcast) requests is supported. This test suite was developed for SIP-over-UDP, although delivery over TCP would simplify the session tear-down after each test case. Preference was given to UDP since earlier versions of SIP (RFC 2543) required only UDP support and left TCP support optional. Consequently, not all implementations available to us supported SIP over TCP.

D. Specifications design

Protocol data unit (PDU) specifications are used as a basis for generating the test cases. The starting point is machine-readable representation of the protocol specification. Our test tool utilizes a custom dialect of the Backus-Naur Form (BNF). An excerpt from the PROTOS BNF to model SIP can be found in Figure 1.

The most commonly used SIP implementations are user agents and proxy server. Both must implement the handling of SIP-INVITE messages. We focus therefore on SIP-INVITE messages in the design of the test material.

Fig. 1. SIP BNF excerpt

```
<SIP-Message> = {
  <Request-Line>
  <Request-Headers>
  <CRLF>
  <Message-Body> #set to SDP-format
}
<Request-Line> = {
  <Method>
  <Sp>
  <Request-URI>
  <Sp>
  <SIP-Version>
  <CRLF>
}
<SIP-Version> = "2.0"
<Method> = "INVITE"
<Sp> = 0x20
```

E. Design of exceptional elements

An exceptional element is an input that has been designed to provoke undesired behavior in the implementation. It can violate the protocol specification, but it is often legal or exploits ambiguities in the specification. In a nutshell, an exceptional element is an input that might not have been considered properly when implementing the software.

Exceptional elements are also modeled as BNF. An example is given in Figure 2. Their design is based on the experience gathered from previous test suites and publicly reported vulnerabilities. Thus, we focus on how well test cases represent the external pressure against robustness rather than on complete coverage of the input space. A test case contains generally one or more exceptional elements, with other elements being unaltered.

Table II lists the used exceptional elements in the test material.

F. Design of test material

The test material consists of test cases simulating hostile input to the implementation. Cases are arranged into 54 test groups, each group covering a certain part of the PDU. Details for the package of 4527 SIP-INVITE test messages

Fig. 2. Exceptional element

```
data <ee-overflow-null> {
    () |
    0x00 #null in front
    0x00 (9x 0x61) |
    0x00 (17x 0x61) |
    0x00 (33x 0x61) |
    0x00 (63x 0x61) |
    0x00 (127x 0x61) |
    0x00 (255x 0x61) |
    0x00 (1024x 0x61) |
    0x00 (16383x 0x61) |
    0x00 (32000x 0x61) |
    0x61 0x00 0x61  | #null in middle
    (9x 0x61) 0x00 (9x 0x61)  |
    (17x 0x61) 0x00 (17x 0x61)  |
    (33x 0x61) 0x00 (33x 0x61)  |
    (63x 0x61) 0x00 (63x 0x61)  |
    (127x 0x61) 0x00 (127x 0x61)  |
    (255x 0x61) 0x00 (255x 0x61)  |
    (1025x 0x61) 0x00 (1024x 0x61)  |
    (16383x 0x61) 0x00 (16385x 0x61)  |
    (32000x 0x61) 0x00 (32000x 0x61)  |
    0x61 0x00 (32767x 0x61) |
    0x61 (2x 0x00) 0x61     | #many nulls
    0x61 (127x 0x00) 0x61   |
    0x61 (1025x 0x00) 0x61  |
    0x61 (2x 0x00) (32767x 0x61) |
    127x  (0x61 0x00) |
    1025x (0x61 0x00) |
    "\" 0x00 |        #some escaped nulls
    "\" 0x00 0x00 |
    1025x ("\" 0x00) |
    1025x ("\" 0x00) 0x00 |
    (63x 0x61) 0x00    | #null in end
    (127x 0x61) 0x00   |
    (255x 0x61) 0x00   |
    (1024x 0x61) 0x00  |
    (16383x 0x61) 0x00 |
    (33000x 0x61) 0x00
}
```

are presented in Table III. The test material was designed to exercise the SIP and SDP headers, their fields and their delimiters in isolation (one-by-one).

G. Injection

PDUs were injected by a simple UDP injector. For test automation, connection teardown to cancel previous INVITE requests had to be implemented. Otherwise, test execution against terminals would have required manual intervention to terminate incoming calls for each test case. Therefore, we send a CANCEL and ACK message after each test case [17, Appendix A].

TABLE II
EXCEPTIONAL ELEMENT CATEGORIES

Name	Description
empty	Omitted (empty) element content
ipv4-ascii	Malformed IPv4 addresses in ASCII and special purpose addresses
overflow-general	“a“ (0x61) character overflows up to 128KB
overflow-slash	Overflows of “/“ up to 128KB
overflow-colon	Overflows of “:“ up to 128KB
overflow-space	Overflows of “ “ up to 128KB
overflow-null	Overflows of 0x61 and nulls (0x00) mixed
overflow-leftbracket	Overflows of “<“ up to 128KB
overflow-rightbracket	Overflows of “>“ up to 128KB
overflow-at	Overflows of “@“ up to 128KB
overflow-equal	Overflows of “=“ up to 128KB
fmtstring	Format strings
utf-8	Malformed UTF-8 sequences
integer-ascii	Pos/Neg ASCII encoded integers
ansi-escape	ANSI escape sequences
sip-version	Malformed ”SIP/2.0”
content-type	Malformed ”application/sdp”
sip-URI	Malformed SIP-URI
sip-tag	Malformed SIP-tags
crlf	Arrangements of CR (0x0d) and LF (0x0a)

The test case design does not account for maximum payload limitations of the transport protocol (64 KBytes minus UDP/IP headers). Thus, almost each group contains a test case that is silently truncated to a configurable limit. This behavior may distort the interpretation of test results, i.e., some of the observed failures may result from the inability to handle truncated packets rather than what is indicated by the applied exceptional element or by the exercised field.

H. Instrumentation

The implementation is monitored for undesirable behavior that could have security implications. Instrumentation methods can be roughly divided into two categories, out-of-band and in-band.

Out-of-band instrumentation on the target platform includes debuggers, resource monitoring or custom-made tools used to extract information from the implementation. This is often the preferred form of instrumentation. For in-band instrumentation, the implementation is monitored via the injection vector, i.e., the same interface used to deliver the test cases. While not checked for protocol conformance, absent or malformed responses can often reveal anomalous conditions such as denial-of-service. Also, the ability to accept subsequent test cases is an indicator of the performance on the previous test case.

For our tests, we use a variant of in-band instrumentation, called valid-case instrumentation. A valid PDU (valid-case), that should result in a valid reply, is sent to the subject between test cases until a response is received. Hence, if no response from the subject is detected, it has failed.

Figure 3 shows an implementation passing a test case. Normally, replies are discarded. There is one exception: after sending the valid-case the injector requires a reply from the implementation before continuing to the next test case. In

Figure 4, the test case 'crashes' the implementation. The subsequent valid-case does not receive a response and the injector resends the valid-case.

I. Results

Results from the test runs are given in Table IV[1]. Results are presented in a tabular form with test cases divided into

[1]Product name and respective vendor of the actual implementation are - in accordance to our policy - omitted.

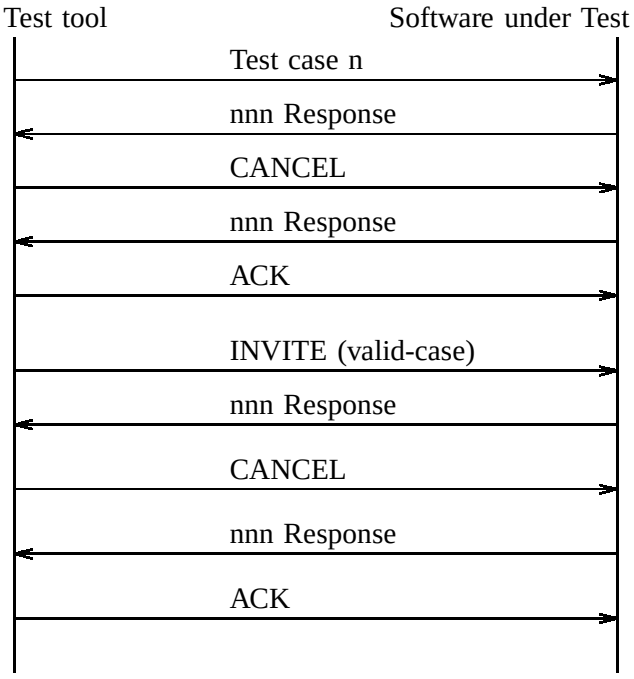
TABLE III
TEST GROUPS

Name	Exceptional Elements	Test Cases
valid	n/a	1
SIP-Method	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-Request-URI	sip-URI	61
SIP-Version	sip-version	75
SIP-Via-Host	ipv4-ascii	106
SIP-Via-Hostcolon	overflow-colon	16
SIP-Via-Hostport	integer-ascii	46
SIP-Via-Version	sip-version	75
SIP-Via-Tag	sip-tag	57
SIP-From-Displayname	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-From-Tag	sip-tag	57
SIP-From-Colon	overflow-colon	16
SIP-From-URI	sip-URI	61
SIP-Contact-Displayname	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-Contact-URI	sip-URI	61
SIP-Contact-Left-Paranthesis	overflow-leftbracket	16
SIP-Contact-Right-Paranthesis	overflow-rightbracket	16
SIP-To	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-To-Left-Paranthesis	overflow-leftbracket	16
SIP-To-Right-Paranthesis	overflow-rightbracket	16
SIP-Call-Id-Value	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-Call-Id-At	overflow-at	16
SIP-Call-Id-Ip	ipv4-ascii	106
SIP-Expires	integer-ascii	46
SIP-Max-Forwards	integer-ascii	46
SIP-Cseq-Integer	integer-ascii	46
SIP-Cseq-String	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SIP-Content-Type	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape, content-type	247
SIP-Content-Length	integer-ascii	46
SIP-Request-CRLF	crlf	10
CRLF-Request	crlf	10
SDP-Attribute-CRLF	crlf	10
SDP-Proto-v-Identifier	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SDP-Proto-v-Equal	overflow-equal	16
SDP-Proto-v-Integer	integer-ascii	46
SDP-Origin-Username	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SDP-Origin-Sessionid	integer-ascii	46
SDP-Origin-Networktype	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SDP-Origin-Ip	overflow-equal	106
SDP-Session	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SDP-Connection-Networktype	overflow-general, overflow-space, overflow-null, utf-8, fmtstring	188
SDP-Connection-Ip	ipv4-ascii	106
SDP-Time-Start	integer-ascii	46
SDP-Time-Stop	empty	1
SDP-Media-Media	overflow-general, overflow-space, overflow-null, fmtstring, utf-8, ansi-escape	193
SDP-Media-Port	integer-ascii	46
SDP-Media-Transport	overflow-general, overflow-space, overflow-null, fmtstring, ansi-escape	118
SDP-Media-Type	integer-ascii	46
SDP-Attribute-Rtpmap	overflow-general, overflow-space, overflow-null, fmtstring, ansi-escape	118
SDP-Attribute-Colon	overflow-colon	16
SDP-Attribute-Payloadtype	integer-ascii	46
SDP-Attribute-Encodingname	integer-ascii	118
SDP-Attribute-Slash	overflow-slash	16
SDP-Attribute-Clockrate	integer-ascii	46

Legend:

- **Name:** of the test group and reflects the header and field name in the protocol specification
- **Exceptional Elements:** integrated exceptional element categories
- **Test Cases:** number of test cases

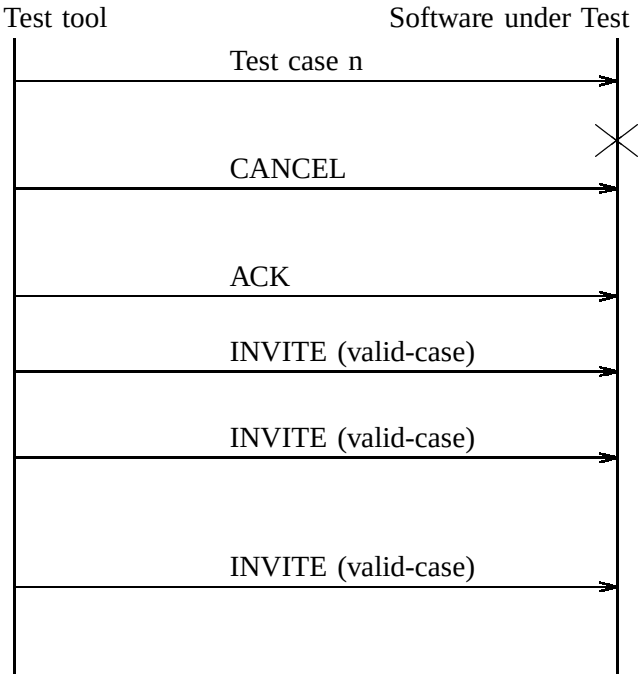
Fig. 3. Passed test case with teardown



Legend:

- **nnn Response:** any response send by the implementation

Fig. 4. Failed test case



test groups. Each failed test case represents, at minimum, a denial-of-service mechanism of exploiting the vulnerability.

TABLE V
RESULTS SUMMARY

Category	Test run	Failed test cases	Failed groups
User Agent	tr-001	N	52
User Agent	tr-002	N	12
User Agent	tr-003	129	9
User Agent	tr-004	0	0
User Agent	tr-005	N	3
User Agent	tr-006	N	45
Proxy	tr-007	N	49
Proxy	tr-008	N	10
Proxy	tr-009	33	4

Legend:

- **N**: we were unable to determine the exact number of failures

In Table IV, the **failed** status is given if any of the following criteria are met and at least one test case in the test group can be identified as being responsible for one of the following failure modes.

- Software under test suffers a fatal failure and stops functioning normally.
- Software crashes or hangs and needs to be restarted manually.
- Software spontaneously restarts itself.
- Software is not respond to a valid message after 16 seconds.

Sometimes, an implementation gets so badly corrupted that there is no way to collect accurate test results for the whole test run. This constitutes a permanent denial-of-service. Untested regions are marked as **unknown**.

Otherwise, the status is **passed**. The results are further summarized in Table V.

Implementation failing in the majority of test groups are likely to have a bug in input parsing common to all header fields, instead of separate bugs for handling each field. We also found that none of the embedded devices passed the test runs.

In most cases, failures represent memory corruption, stack corruption or other fatal error conditions. Some of these may lead to exposure to typical buffer overflow exploits, allowing running of arbitrary code [3] or modification of the target system.

J. Limitations

Our approach has limitations. The principal limitations of software testing, as pointed out by Dijkstra, that ‘Program testing can be used to show the presence of bugs, but never their absence’ [18], exist, our tool covers just a miniscule portion of the input space. The provided instrumentation leaves open questions about the actual bugs behind the observed failure modes.

III. REPORTING PROCESS

We applied the *constructive vulnerability disclosure* model as proposed in [19] to handle this multi-vendor, multi-vulnerability case. We aimed to distribute the test suite and inform vendors on this issue. Communication with all parties

was done via email. We describe the different phases during the release of the test suite and discuss the process.

A. Development phase

We described this phase in the previous section. During this phase internal testing was conducted against available products and the test material and documentation were prepared.

B. Pre-release phase

Next, the Computer Emergency Response Team / Coordination Center (CERT/CC), acting as the coordinator in handling this vulnerability case was informed of our test

TABLE IV
OBSERVED FAILURES FOR TEST GROUPS

	Implementation								
test group	**001**	**002**	**003**	**004**	**005**	**006**	**007**	**008**	**009**
valid	-	-	-	-	-	-	-	-	-
SIP-Method	X	-	-	-	-	X	-	-	-
SIP-Request-URI	X	-	-	-	-	X	-	X	-
SIP-Version	X	-	-	-	-	X	-	-	-
SIP-Via-Host	X	-	X	-	X	X	X	X	-
SIP-Via-Hostcolon	X	-	X	-	-	-	-	-	-
SIP-Via-Hostport	X	X	X	-	-	X	-	-	-
SIP-Via-Version	X	-	X	-	-	X	-	-	-
SIP-Via-Tag	X	?	?	-	-	X	-	-	-
SIP-From-Displayname	X	X	X	-	-	X	-	-	X
SIP-From-Tag	X	X	-	-	-	X	-	-	-
SIP-From-Colon	X	X	-	-	-	X	-	X	X
SIP-From-URI	X	X	X	-	X	X	-	X	-
SIP-Contact-Displayname	X	-	-	-	-	X	-	-	-
SIP-Contact-URI	X	-	-	-	-	X	-	X	-
SIP-Contact-Left-Paranthesis	X	-	-	-	-	X	-	-	-
SIP-Contact-Right-Paranthesis	X	-	-	-	-	X	-	-	-
SIP-To	X	-	-	-	-	X	X	X	-
SIP-To-Left-Paranthesis	X	-	-	-	-	X	-	-	X
SIP-To-Right-Paranthesis	X	-	-	-	-	X	-	-	X
SIP-Call-Id-Value	X	X	?	-	-	X	-	X	-
SIP-Call-Id-At	X	X	-	-	-	X	-	-	-
SIP-Call-Id-Ip	X	X	-	-	-	X	X	X	-
SIP-Expires	X	-	-	-	-	X	-	-	-
SIP-Max-Forwards	X	X	?	-	-	X	-	-	-
SIP-Cseq-Integer	X	-	-	-	-	X	-	-	-
SIP-Cseq-String	X	X	X	-	-	X	-	-	-
SIP-Content-Type	X	-	-	-	-	X	-	-	-
SIP-Content-Length	X	X	-	-	-	X	-	X	-
SIP-Request-CRLF	X	-	-	-	X	-	-	X	-
CRLF-Request	X	-	-	-	-	-	-	-	-
SDP-Attribute-CRLF	X	-	-	-	-	-	-	-	-
SDP-Proto-v-Identifier	X	-	X	-	-	X	-	-	-
SDP-Proto-v-Equal	X	-	-	-	-	-	-	-	-
SDP-Proto-v-Integer	X	-	-	-	-	X	-	-	-
SDP-Origin-Username	X	X	-	-	-	X	-	-	-
SDP-Origin-Sessionid	X	-	-	-	-	X	-	-	-
SDP-Origin-Networktype	X	-	-	-	-	X	-	-	-
SDP-Origin-Ip	X	-	X	-	-	X	X	-	-
SDP-Session	X	-	-	-	-	X	-	-	-
SDP-Connection-Networktype	X	-	-	-	-	X	-	-	-
SDP-Connection-Ip	X	-	-	-	-	X	X	-	-
SDP-Time-Start	X	-	-	-	-	X	-	-	-
SDP-Time-Stop	-	-	-	-	-	-	-	-	-
SDP-Media-Media	X	-	-	-	-	X	-	-	-
SDP-Media-Port	X	-	-	-	-	X	-	-	-
SDP-Media-Transport	X	-	-	-	-	X	-	-	-
SDP-Media-Type	X	-	-	-	-	X	-	-	-
SDP-Attribute-Rtpmap	X	-	-	-	-	X	-	-	-
SDP-Attribute-Colon	X	-	-	-	-	-	-	-	-
SDP-Attribute-Payloadtype	X	-	-	-	-	X	-	-	-
SDP-Attribute-Encodingname	X	-	-	-	-	X	-	-	-
SDP-Attribute-Slash	X	-	-	-	-	-	-	-	-
SDP-Attribute-Clockrate	X	-	-	-	-	X	-	-	-

Legend:

- **X**: failed
- **-**: passed
- **?**: unknown

results. The coordinator acts as a neutral third party between us, the originator, and the affected vendors. The coordinator distributed the material to individual vendors. On our part the access to the material was limited to these parties.

C. Release phase

After a grace period of 121 days, the suite was released to the public [17]. The release followed the advisory CA-2003-06 [20] by CERT/CC on this vulnerability.

IV. Discussion

The feedback on the test suite and the release was positive and the test suite achieved a fair amount of public and community interest. During the communication with vendors we got the impression that the methodology used was not widely known. While this paper describes the use of robustness testing for SIP, the method applies to all network protocols. Some vendors had not been contacted by CERT/CC before, which added extra value to the communication process as a preparatory exercise.

We visited the interoperability testing SIPit11 and discussed our approach with participants. The test suite was not released at that time, we distributed the test suite via the coordinator CERT/CC. We received a report of some vendors believing that we had released test results to the public, which never happened [21]. Our intention was to included as many participants as possible during the pre-release phase, which gave early access to the test suite.

CERT/CC lists 92 vendors in their vulnerability note [22]. 51 (55%) have not provided a vendor statement, 32 vendors (35%) said that they are not vulnerable and 9 vendors (10%) claimed to be vulnerable to the test suite. The high rate of non-responders was in line with similar cases like SNMP [23] (53%). The high rate of invulnerable vendors appears to be mostly due to CERT/CC's practice of listing all contacted vendors, not just vendors with SIP products. Therefore, we see no contradiction with our vulnerability rate of 88% gathered during the development phase.

Some of the vendors prepared patches during the pre-release phase. Some reacted late, or not at all.

V. Conclusion

We applied the PROTOS approach to SIP. The systematic testing for implementation level vulnerabilities gives vendors the possibility to harden their implementations against this kind of attacks. Only one from the sample of nine implementations survived the test material as it is. Vendors were informed during the prerelease phase. After a grace period the material was made publicly available. By applying this approach we hope to give implementors and their customers a tool to improve the robustness of SIP implementations.

Acknowledgment

We wish to express our gratitude to individual vendors who worked with us to protect their customers. We are grateful to CERT/CC for their patient help, advice and active role during the vulnerability process.

References

[1] (2001-) SANS security alert consensus - archive. [Online]. Available: http://www.sans.org/newsletters/sac/

[2] (2002) CERT/CC statistics 1988-2002. cert_stats.html. [Online]. Available: http://www.cert.org/stats/

[3] Aleph One. (1996, Nov.) Smashing the stack for fun and profit. BugTraq and r00t and Underground.Org. [Online]. Available: http://www.shmoo.com/phrack/Phrack49/p49-14

[4] scut. (2001, Sept.) Exploiting format string vulnerabilities. formatstring-1.2.tar.gz. team teso. [Online]. Available: http://teso.scene.at/releases/

[5] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: session initiation protocol," RFC 3261, June 2002.

[6] (2003) 3GPP home page. [Online]. Available: http://www.3gpp.org/

[7] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle, "SIPstone - benchmarking SIP server performance," sipstone_0402.pdf, Apr. 2002. [Online]. Available: http://www.sipstone.org/files/

[8] "SiPit - SIP interoperability test event," 2003. [Online]. Available: http://www.sipit.net/

[9] A. Johnston, J. Rosenberg, and H. Schulzrinne. (2002, August) Session Initiation Protocol Torture Test Messages. IETF Draft.

[10] (2003) The SIP center - testing and measurement. vsts_testing.html. [Online]. Available: http://www.sipcenter.com/vsts/

[11] M. Ranganathan, O. Deruelle, and D. Montgomery. (2002, Oct.) Testing SIP call flows using XML protocol templates. 100 Bureau Drive, Gaithersburg, MD 20899, USA.

[12] (2003) Using TTCN to test VoIP. ptccsip_osp.htm. [Online]. Available: http://www.etsi.org/ptcc/

[13] PROTOS - security testing of protocol implementations. [Online]. Available: http://www.ee.oulu.fi/research/ouspg/protos/

[14] R. Kaksonen, *A Functional Method for Assessing Protocol Implementation Security*. VTT Publication series, 2001. [Online]. Available: http://www.inf.vtt.fi/pdf/publications/2001/

[15] B. Beizer, *Software Testing Techniques*, 2nd ed., 1990.

[16] M. Handley and V. Jacobson. (1998, April) SDP: session description protocol. RFC 2327.

[17] "PROTOS test-suite: c07-sip." [Online]. Available: http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/

[18] E. W. Dijkstra, "Structured programming," Aug. 1969, circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd02xx/

[19] M. Laakso, A. Takanen, and J. Röning, "Introducing constructive vulnerability disclosures," 2001. [Online]. Available: http://www.ee.oulu.fi/research/ouspg/protos/sota/FIRST2001-disclosures/

[20] CERT/CC. (2003, February) CERT advisory CA-2003-06 multiple vulnerabilities in implementations of the session initiation protocol (SIP). CA-2003-06.html. [Online]. Available: http://www.cert.org/advisories/

[21] R. Sparks, private communication, Nov. 2002.

[22] CERT/CC. (2003, February) Vulnerability note VU#528719 - multiple implementations of the session initiation protocol (SIP) contain vulnerabilities. [Online]. Available: http://www.kb.cert.org/vuls/id/528719

[23] "PROTOS test-suite: c06-snmpv1." [Online]. Available: http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/