Let's Parse to Prevent Pwnage

Invited position paper

Mike Samuel
Google Inc.

Úlfar Erlingsson
Google Inc.

Abstract

Software that processes rich content suffers from endemic security vulnerabilities. Frequently, these bugs are due to *data confusion*: discrepancies in how content data is parsed, composed, and otherwise processed by different applications, frameworks, and language runtimes. Data confusion often enables code injection attacks, such as cross-site scripting or SQL injection, by leading to incorrect assumptions about the encodings and checks applied to rich content of uncertain provenance. However, even for well-structured, value-only content, data confusion can critically impact security, e.g., as shown by XML signature vulnerabilities [12].

This paper advocates the position that data confusion can be effectively prevented through the use of simple mechanisms—based on parsing—that eliminate ambiguities by fully resolving content data to normalized, clearly-understood forms.

Using code injection on the Web as our motivation, we make the case that automatic defense mechanisms should be integrated with programming languages, application frameworks, and runtime libraries, and applied with little, or no, developer intervention. We outline a scalable, sustainable approach for developing and maintaining those mechanisms. The resulting tools can offer comprehensive protection against data confusion, even when multiple types of rich content data are processed and composed in complex ways.

1 Data Confusion and Why Parsing Helps

A persistent source of security issues is *data confusion*: vulnerabilities caused by inconsistencies between different software in the parsing, composition, and overall processing of rich content. Data confusion has already led to large-scale exploits such as rapidly-spreading Web application worms [18], and its risk is increasing, with the growth of distributed and cloud computing.

Examples of data confusion have arisen in the handling of nested HTML tags [8], apostrophes in SQL statements [19], signature scopes in XML protocol messages [12], and encoded length fields in binary data [9].

Data confusion cannot be eliminated simply by training software developers or by exhorting them to be more careful. For general-purpose software, data is usually of uncertain provenance and, locally, it is usually hard to tell what data can be trusted, what data properties have been checked, and what assumptions about data are made elsewhere. Even if all software for processing rich content was written with the utmost care—and developers had the right incentives, know-how, and resources—discrepancies between different developers' decisions would still be sure to introduce vulnerabilities.

On the other hand, to avoid data confusion, it is often sufficient to simply *normalize* the content data by parsing and re-serializing the data. Normalization has been previously used by security mechanisms, e.g., to eliminate TCP fragmentation ambiguities [22] and to build deterministic HTML parse trees [21]. It benefits security by resolving ambiguities, by simplifying the data encoding (e.g., via conversion), and by eliding deprecated aspects or unnecessary functionality from the content.

For example, to display raster images, only a single (compressed) encoding and color space (e.g., sRGB) is strictly necessary. Thus, by normalizing to a single form of bitmap data, most of the attack surface due to the variety of image formats (and all of their myriad encodings and options) can be eliminated. Notably, such normalization can benefit even the security of legacy software: eliminating esoteric options and encodings will prevent most known JPEG and PNG exploits (e.g., [1, 9]).

Clearly, automatic mechanisms based on trustworthy parsing can prevent many types of data confusion by reducing the attack surface due to the divergent assumptions of different software.

Centralized, trustworthy parsing can be helpful in other ways, as well. For example, such parsing could

support large-scale collection of statistics about content data that would help identify corner cases and rarely-used features—both a common source of vulnerabilities. Also, such processing could ensure that content data met the required constraints of certain, preferred software—such as that deemed to be standard, or most secure—and thereby eliminate further sources of data confusion, such as those underlying recently-discovered attacks on anti-virus scanners [11].

Centralized normalization could even improve performance, and eliminate redundant work, by serializing content data to a new unambiguous, highly-efficient structured format (e.g., based on Google's Protocol Buffers [7]), instead of back to the original data format. In the context of the Web, Michal Zalewski of Google has pointed out many ancillary benefits of similar new formats, such as reduced latency of loading Web pages.

2 Towards Comprehensive Defenses

Unfortunately, to overcome endemic data confusion, simple centralized mechanisms are not sufficient. Rich content may be composed and processed on both clients and servers and typically embeds some form of executable code—and that code often encodes complex predicates and content introspection that prevents static reasoning about behavior. During such processing, data confusion can easily result in *code injection* vulnerabilities, where attacker-controlled characters are included as part of executed expressions, in unexpected contexts [14]. Therefore, it is not surprising that, for many years, the most commonly-reported security vulnerabilities have been SQL-injection and Cross-Site Scripting (XSS) in Web applications [2, 6].

The remainder of this position paper uses the context of Web applications to outline a sustainable approach for developing comprehensive protections against data confusion—even when multiple types of rich content data are processed and composed in complex ways. Those defenses are based on the close integration of automated mechanisms for content data normalization, sanitization, and templating, as well as execution sandboxing, into client and server Web programming languages. For scalability, we describe how those mechanisms can be based on annotated parse-tree grammars developed independently of any language, platform, or application.

2.1 The Case of HTML and the Web

Web application developers are forced, by necessity, to restrict their attention to functionality that works reliably cross-platform (e.g., "JavaScript: The Good Parts" [4]). On the other hand, attackers can make full use of all the Web's bad parts: its corner cases, esoteric platform-specific features, and poorly-thought-out functionality. Also, the recent fast-paced experimentation with new features, languages, and application frameworks for the Web and for cloud computing forces defenders to consider an impossible menagerie of technologies: ASP.NET, CoffeeScript, Ruby on Rails, Django, jQuery, JSF, Dart, and Go—to name but a handful.

	Untrusted data	**Untrusted code**
Untrusted ctx.	Lowering	Sanitization
Trusted ctx.	Safe templating	Sandboxing

Figure 1: Techniques for securely handling Web content data, across different processing contexts and input data.

Security-savvy Web developers must know how to (manually) employ a range of ad hoc tools for securely composing content strings from untrusted and trusted sources. In particular, consistent use of tools like SQL prepared statements or auto-escaped HTML templates in Web application frameworks can greatly reduce susceptibility to data confusion [5]. More principled, safe-by-construction mechanisms (such as those in [19, 20]) have seen little adoption, since they have required extensive modification of the Web application source code as well as substantial programmer retraining.

These existing tools fall on two axes, as depicted in Figure 1. The first axis is determined by the initial runtime processing of attacker-controlled inputs: *untrusted data* will be encoded into strings, whereas *untrusted code* will be passed to a language interpreter.

```
untrusted = x; // is "javascript:..." ?
location = untrusted + '?foo=bar';
```

For example, the above code fragment composes untrusted data with a trusted literal, '`?foo=bar`', to form a location URL. Here, the application developer may have failed to check that the untrusted data encodes a URL domain path, thereby enabling an attack.

By contrast, untrusted code may exercise more authority than the Web application developer intends.

```
Dear Sir,<script>IPwnYou()</script>
```

For example, a Web mail client would be wise to remove the "`<script>`" from the above HTML email body.

The second axis depends on whether the Web application itself is trustworthy. *Untrusted contexts* of processing allow attacker-controlled input to fully determine the rendering of content data. However, more often Web servers or client browsers may process attacker-controlled inputs in a *trusted context*—e.g., to insert untrusted data or code into holes in templates trusted by the Web application. For example, in PHP, a Web application might use a template "`<b>$untrusted</b>`" with trusted HTML tokens "`<b>`" and "`</b>`".

3 Building Sustainable Defenses at Scale

Annotated, high-level parse-tree grammar specifications can be used to drive data confusion defenses like normalization, encoding, sanitization, and templating. Such annotated grammars can be developed and maintained by security-minded engineers and pen-testers familiar with the content format (such as HTML and CSS)—independent of any platform—and, on each platform, compiled to make use of the correct specific primitives for the secure processing of content data.

Seperately, the underlying platform-specific content security primitives can be provided as part of runtime platforms and application frameworks (such as Python and ASP.NET). Thus, the cost of defenses can be linear—$O(c+p)$, and not $O(c \cdot p)$, for c content data formats and p platforms—which, in particular, can allow code injection defenses to be sustainably scaled to all the Web's different languages, frameworks, and platforms.

Figure 1 shows the techniques that might be applied by platform-specific content security primitives. When deployed as automatic mechanisms, driven by annotated grammars, these techniques based on parsing and normalization can comprehensively prevent data confusion—as long as they are fully integrated into Web application languages, frameworks, and libraries.

The first technique, *encoded lowering*, normalizes content to a plain-data format that elides possible control characters, and is free of rich features—e.g., "1<2" becomes "1<2" in HTML [10]. The second, *sanitization*, normalizes content by eliding all features not deemed to be in a safe subset. Sanitizers like HTML Purifier [23] allow web-sites to inline untrusted HTML. The third, *auto-escaped templating*, composes untrusted content with trusted literals, as specified in a structured fashion in the Web application—e.g., as is done in Google's Closure Templates [16]. Importantly, this composition must guarantee that untrusted content cannot have side effects, and that the intended semantics of trusted literals are preserved. Finally, *sandboxing*, like PHP's Runkit Sandbox [13], contains the effects of untrusted content by limiting the interface available during its processing.

3.1 Basing Tools on Annotated Grammars

Grammars are widely used for specifying data formats and language parsers, by defining both the set of language literals and the structure of a parse tree (based on the grammar productions). However, traditional grammars do not specify aspects necessary to fight data confusion. In the context of the Web, such aspects include:

- The relationship between data values and substrings in the language; e.g., the JSON grammar does not specify that \' in "I\'m" encodes an apostrophe.
- The grammar for encoded strings; e.g., that the attribute in <a href="?a=b&c=d"> is encoding the URL ?a=b&c=d.
- A mapping from strings to "safer" strings; e.g., that removing JavaScript and external URLs makes HTML content safer.
- Resolution of references within content data; e.g., to uniquely map identifiers to DOM elements.

Grammars for languages and data formats can be annotated to provide extra information about language substrings and units of content data—much like how the C# and Java programming languages allow annotations [3].

```
HTML     := (text | link)*
text     := ([^&<"] | entity)*
entity   := "&amp;" | "&lt;" | ...
link     := "<a" href? ">" text "</a>"
href     := " href=\"" text "\""

URL      := ext_url | js_url | ...
ext_url  := "https://" uri_char*
js_url   := "javascript:" uri_char*
uri_char := [^{}\%] | "\%" hex hex
```

For example, consider the above BNF grammar for a small subset of HTML and URLs, which does not allow tools to infer that

- < encodes a "<",
- the quoted attribute encodes a URL using entities,
- js_url content data encodes JavaScript,
- the hex digits in "%" hex hex encode a byte,
- or, that code may follow the javascript prefix.

```
href   := " href="
          @Embed{URL}('"' text '"')
entity := (@Char{"&"}("&")) | ...
```

On the other hand, a grammar where href and entity are annotated, as shown above, can allow tools to infer

- that a known-safe URL @Embeds as a href attribute value after converting & to &, etc.,
- how to encode a plain-text value to a URL, and, how to sanitize a URL to a safe URL.

Annotated grammars can be converted to pushdown automata, as in Figure 2. Subsequently, each such automaton can be compiled into code that performs encoded lowering by converting untrusted content data into a series of automaton inputs—for example, (start list, start string, char 65, end string, end char)—and searching the automaton for states that satisfy that input series. Such searches can be efficient, if automata compilation pre-computes possible input paths and makes use of the control-flow constructs for the Web application platform. Similarly, efficient code for content sanitizers and the

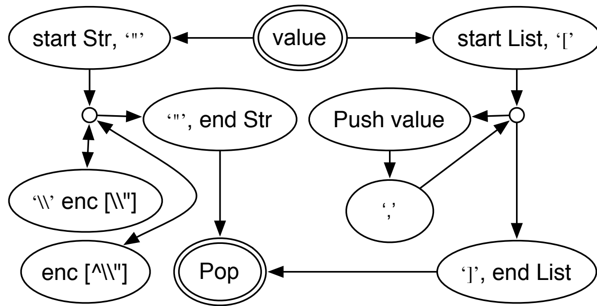

Figure 2: Pushdown automaton to parse a JSON subset.

context-propagation in templating systems can also be compiled from grammar-derived automata.

For testing, we can exploit the division of labor between maintainers of content data grammars and the platform-specific normalization primitives to achieve test coverage. Grammar maintainers can maintain test suites for the encoders, sanitizers, and context propagators derived from their grammars. These grammar-based tests can be run directly against the pushdown automata derived from the grammars, and existing tools for fuzzing untrusted content data can be used to stress test grammars. Independently, platform-specific maintainers can fuzz the grammars, to stress their code generation, and also use the tests of expected automata behavior to verify the consistency of the code they generated for normalization primitives.

3.2 Language and Runtime Integration

Even if mechanisms to defend against code injection were ubiquitous—deployed and fully supported on all platforms—Web application developers would continue to use code like "$untrusted_input", which, in PHP, is succinct, easy to write, and also unsafe. Thus, we propose that data confusion defenses should be integrated closely with programming languages: "$untrusted_input" should do what programmers intend when they write it. Integration with programming languages can be facilitated by delaying content data operations until the processing context can be safely determined, whether by overloading operators, "(re)defining syntactic sugar" [17], or using reflective mechanisms in legacy languages [15].

4 Conclusions

Data confusion vulnerabilities stem from differences between the meaning of rich content data, as understood by different software applications. On the Web, and in cloud computing, data confusion may be impossible to eradicate, due to the plethora of rich content formats coupled with the multiplicity of different computers, platforms, frameworks, languages, libraries, and runtimes upon which applications are built. However, as this position paper outlines, the development and maintenance of data confusion defenses can be sustainable even at the scale required to cover nearly all Web technologies. Let's build and deploy such defenses, and, instead of giving up, let's parse to prevent pwnage.

References

[1] AEDLA, J. libpng: code execution, 2012. http://lwn.net/Articles/481976/.

[2] BOBERSKI, M., ET AL. The ten most critical Web application security risks. Tech. rep., OWASP, 2010.

[3] BRACHA, G., ET AL. *JSR 175: A metadata facility for the Java programming language*. JCP, 2004.

[4] CROCKFORD, D. *JavaScript: The Good Parts*. O'Reilly, 2008.

[5] FINIFTER, M., AND WAGNER, D. Exploring the relationship between Web application development tools and security. In *USENIX conference on Web application development* (2011).

[6] FONSECA, J., VIEIRA, M., AND MADEIRA, H. Testing and comparing Web vulnerability scanning tools for SQL injection and XSS attacks. In *IEEE PRDC* (2007).

[7] GOOGLE. Protocol buffers, 2012. http://code.google.com/p/protobuf/.

[8] HICKSON, I. Serializing HTML fragments: Warning!, 2012. http://www.w3.org/TR/html5/the-end.html.

[9] HORNAT, C. JPEG vulnerability: A day in the life of the JPEG vulnerability, 2004. http://infosecwriters.com/text_resources/pdf/JPEG.pdf.

[10] INCHOWSKI, J. OWASP Java encoder project. http://code.google.com/p/owasp-java-encoder/.

[11] JANA, S., AND SHMATIKOV, V. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security and Privacy* (2012).

[12] MCINTOSH, M., AND AUSTEL, P. XML signature element wrapping attacks and countermeasures. In *ACM Workshop on Secure Web Services* (2005).

[13] PHP.NET. *RunKit_Sandbox*, 2012. http://php.net/manual/en/runkit.sandbox.php.

[14] RAY, D., AND LIGATTI, J. Defining code-injection attacks. In *ACM POPL* (2012).

[15] SAMUEL, M. Secure string interpolation. http://tinyurl.com/secinterp, 2008.

[16] SAMUEL, M. Closure tools security, 2012. http://tinyurl.com/closure-sec.

[17] SAMUEL, M., ET AL. *EcmaScript Quasi-Literals*. EcmaScript TC39, 2011.

[18] SAMY. The Samy worm. http://namb.la/popular.

[19] SU, Z., AND WASSERMANN, G. The essence of command injection attacks in Web applications. In *ACM POPL* (2006).

[20] TALAGA, P., AND CHAPIN, S. Towards a guaranteed (X)HTML compliant dynamic Web application. Springer LNBIP 101, 2012.

[21] TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *IEEE Symposium on Security and Privacy* (2009).

[22] VUTUKURU, M., BALAKRISHNAN, H., AND PAXSON, V. Efficient and robust TCP stream normalization. In *IEEE Symposium on Security and Privacy* (2008).

[23] YANG, E. Z. Standards-compliant HTML filtering, 2012. http://htmlpurifier.org.