Secure Content Sniffing for Web Browsers, *or* How to Stop Papers from Reviewing Themselves

Adam Barth
UC Berkeley

Juan Caballero
UC Berkeley and CMU

Dawn Song
UC Berkeley

Abstract

Cross-site scripting defenses often focus on HTML documents, neglecting attacks involving the browser's content-sniffing algorithm, which can treat non-HTML content as HTML. Web applications, such as the one that manages this conference, must defend themselves against these attacks or risk authors uploading malicious papers that automatically submit stellar self-reviews. In this paper, we formulate content-sniffing XSS attacks and defenses. We study content-sniffing XSS attacks systematically by constructing high-fidelity models of the content-sniffing algorithms used by four major browsers. We compare these models with Web site content filtering policies to construct attacks. To defend against these attacks, we propose and implement a principled content-sniffing algorithm that provides security while maintaining compatibility. Our principles have been adopted, in part, by Internet Explorer 8 and, in full, by Google Chrome and the HTML 5 working group.

1. Introduction

For compatibility, every Web browser employs a *content-sniffing algorithm* that inspects the contents of HTTP responses and occasionally overrides the MIME type provided by the server. For example, these algorithms let browsers render the approximately 1% of HTTP responses that lack a `Content-Type` header. In a competitive browser market, a browser that guesses the "correct" MIME type is more appealing to users than a browser that fails to render these sites. Once one browser vendor implements content sniffing, the other browser vendors are forced to follow suit or risk losing market share [1].

If not carefully designed for security, a content-sniffing algorithm can be leveraged by an attacker to launch cross-site scripting (XSS) attacks. In this paper, we study these *content-sniffing XSS attacks*. Aided by a technique we call *string-enhanced white-box exploration*, we extract models of the content-sniffing algorithms used by four major browsers and use these models to find content-sniffing XSS attacks that affect Wikipedia, a popular user-edited encyclopedia, and HotCRP, the conference management Web application used by the 2009 IEEE Privacy & Security Symposium. We then propose fixing the root cause of these vulnerabilities: the browser content-sniffing algorithm. We design an algorithm based on two principles and evaluate the compatibility of our algorithm on over a billion HTTP responses.

```
%!PS-Adobe-2.0
%%Creator: <script> ... </script>
%%Title: attack.dvi
```

Figure 1. A chameleon PostScript document that Internet Explorer 7 treats as HTML.

Attacks. We illustrate content-sniffing XSS attacks by describing an attack against the HotCRP conference management system. Suppose a malicious author uploads a paper to HotCRP in PostScript format. By carefully crafting the paper, the author can create a *chameleon* document that both is valid PostScript and contains HTML (see Figure 1). HotCRP accepts the chameleon document as PostScript, but when a reviewer attempts to read the paper using Internet Explorer 7, the browser's content-sniffing algorithm treats the chameleon as HTML, letting the attacker run a malicious script in HotCRP's security origin. The attacker's script can perform actions on behalf of the reviewer, such as giving the paper a glowing review and a high score.

Although content-sniffing XSS attacks have been known for some time [2]–[4], the underlying vulnerabilities, discrepancies between browser and Web site algorithms for classifying the MIME type of content, are poorly understood. To illuminate these algorithms, we build detailed models of the content-sniffing algorithms used by four popular browsers: Internet Explorer 7, Firefox 3, Safari 3.1, and Google Chrome. For Firefox 3 and Google Chrome, we extract the model using manual analysis of the source code. For Internet Explorer 7 and Safari 3.1, which use proprietary content-sniffing algorithms, we extract the model of the algorithm using string-enhanced white-box exploration on their binaries. This white-box exploration technique reasons directly about strings and generates models for closed-source algorithms that are more accurate than those generated using black-box approaches. Using our models, we find such a discrepancy in Wikipedia, leading to a content-sniffing XSS attack (see Figure 2) that eluded Wikipedia's developers.

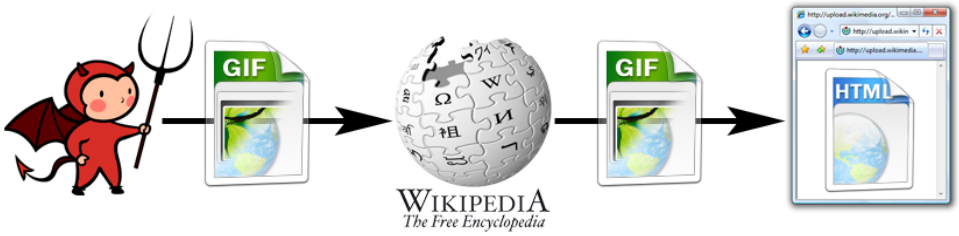

Figure 2. To mount a content-sniffing XSS attack, the attacker uploads a GIF/HTML chameleon to Wikipedia. The browser treats the chameleon as HTML and runs the attacker's JavaScript.

Defenses. Although Web sites can use our models to construct a correct upload filter today, we propose fixing the root cause of content-sniffing XSS attacks by changing the browser's content-sniffing algorithm. To evaluate the security properties of our algorithm, we introduce a threat model for content-sniffing XSS attacks, and we suggest two design principles for a secure content-sniffing algorithm: *avoid privilege escalation*, which protects sites that limit the MIME types they use when serving malicious content, and *use prefix-disjoint signatures*, which protects sites that filter uploads. We evaluate the deployability of our algorithm using Google's search index and opt-in user metrics from Google Chrome users. Using metrics from users who have opted in, we improve our algorithm's security by removing over half of the algorithm's MIME signatures while retaining 99.996% compatibility with the previous version of the algorithm.

Google has deployed our secure content-sniffing algorithm to all users of Google Chrome. The HTML 5 working group has adopted our secure content-sniffing principles in the draft HTML 5 specification [5]. Microsoft has also partially adopted one of our principles in Internet Explorer 8. We look forward to continuing to work with browser vendors to improve the security of their content-sniffing algorithms and to eliminate content-sniffing XSS attacks.

Contributions. We make the following contributions:

- We build high-fidelity models of the content-sniffing algorithms of Internet Explorer 7, Firefox 3, Safari 3.1, and Google Chrome. To extract models from the closed-source browsers, we use string-enhanced white-box exploration on the binaries.
- We use these models to craft attacks against Web sites and to construct a comprehensive upload filter these sites can use to defend themselves.
- We propose two design principles for secure content-sniffing algorithms and evaluate the security and compatibility of these principles using real-world data.
- We implement and deploy a content-sniffing algorithm based on our principles in Google Chrome and report adoption of our principles by standard bodies and other browser vendors.

Organization. Section 2 describes our analysis techniques, the content-sniffing algorithms used by four major browsers, and the concrete attacks we discover. Section 3 presents our threat model, a server-based filtering defense, our two principles for secure content sniffing, a security analysis of our principles, and a compatibility analysis of our implementation. Section 4 discusses related work. Section 5 concludes.

2. Attacks

In this section, we study content-sniffing XSS attacks. First, we provide some background information. Then, we introduce content-sniffing XSS attacks. Next, we describe a technique for constructing models from binaries and apply that technique to extract models of the content-sniffing algorithm from four major browsers. Finally, we construct attacks against two popular Web sites by comparing their upload filters with our models.

2.1. Background

In this section, we provide background information about how servers identify the type of content included in an HTTP response. We do this in the context of a Web site that allows its users to upload content that can later be downloaded by other users, such as in a photograph sharing or a conference management site.

Content-Type. HTTP identifies the type of content in uploads or downloads using the `Content-Type` header. This header contains a *MIME type*[1] such as `text/plain` or `application/postscript`. When a user uploads a file using HTTP, the server typically stores both the file itself and a MIME type. Later, when another user requests the file, the Web server sends the stored MIME type in the `Content-Type` header. The browser uses this MIME type to determine how to present the file to the user or to select an appropriate plug-in.

1. Multipurpose Internet Mail Extensions (MIME) is an Internet standard [6]–[8] originally developed to let email include non-text attachments, text using non-ASCII encodings, and multiple pieces of content in the same message. MIME defines MIME types, which are used by a number of protocols, including HTTP.

Some Web servers (including old versions of Apache [9]) send the wrong MIME type in the `Content-Type` header. For example, a server might send a GIF image with a `Content-Type` of `text/html` or `text/plain`. Some HTTP responses lack a `Content-Type` header entirely or contain an invalid MIME type, such as `*/*` or `unknown/unknown`. To render these Web sites correctly, browsers use *content-sniffing* algorithms that guess the "correct" MIME type by inspecting the contents of HTTP responses.

Upload filters. When a user uploads a file to a Web site, the site has three options for assigning a MIME type to the content: (1) the Web site can use the MIME type received in the `Content-Type` header; (2) the Web site can infer the MIME type from the file's extension; (3) the Web site can examine the contents of the file. In practice, the MIME type in the `Content-Type` header or inferred from the extension is often incorrect. Moreover, if the user is malicious, neither option (1) nor option (2) is reliable. For these reasons, many sites choose option (3).

2.2. Content-Sniffing XSS Attacks

When a Web site's upload filter differs from a browser's content-sniffing algorithm, an attacker can often mount a *content-sniffing XSS attack*. In a content-sniffing XSS attack, the attacker uploads a seemingly benign file to an honest Web site. Many Web sites accept user uploads. For example, photograph sharing sites accept user-uploaded images and conference management sites accepts user-uploaded research papers. After the attacker uploads a malicious file, the attacker directs the user to view the file. Instead of treating the file as an image or a research paper, the user's browser treats the file as HTML because the browser's content-sniffing algorithm overrides the server's MIME type. The browser then renders the attacker's HTML in the honest site's security origin, letting the attacker steal the user's password or transact on behalf of the user.

To mount a content-sniffing XSS attack, the attacker must craft a file that will be accepted by the honest site and be treated as HTML by the user's browser. Crafting such a file requires exploiting a mismatch between the site's upload filters and the browser's content-sniffing algorithm. A *chameleon* document is a file that both conforms to a benign file format (such as PostScript) and contains HTML. Most file formats admit chameleon documents because they contain fields for comments or metadata (such as EXIF [10]). Site upload filters typically classify documents into different MIME types and then check whether that MIME type belongs to the site's list of allowed MIME types. These sites typically accept chameleon documents because they are formated correctly. The browser, however, often treats a well-crafted chameleon as HTML.

The existence of chameleon documents has been known for some time [2]. Recently, security researchers have suggested using PNG and PDF chameleon documents to launch XSS attacks [3], [4], [11], [12], but these researchers have not determined which MIME types are vulnerable to attack, which browsers are affected, or whether existing defenses actually protect sites.

2.3. Model Extraction

We investigate content-sniffing XSS attacks by extracting high-fidelity models of content-sniffing algorithms from browsers and Web sites. When source code is available, we manually analyze the source code to build the model. Specifically, we manually extract models of the content-sniffing algorithms from the source code of two browsers, Firefox 3 and Google Chrome, and the upload filter of two Web sites, Wikipedia [13] and HotCRP [14].

Extracting models from Internet Explorer 7 and Safari 3.1[2] is more difficult because their source code is not available publicly. We could use black-box testing to construct models by observing the outputs generated from selected inputs, but models extracted by black-box testing are often insufficiently accurate for our purpose. For example, the Wine project [15] used black-box testing and documentation [16] to re-implement Internet Explorer's content-sniffing algorithm, but Wine's content-sniffing algorithm differs significantly from Internet Explorer's content-sniffing algorithm. For example, the Wine signature for HTML contains just the `<html` tag instead of the 10 tags we find in Internet Explorer's content-sniffing algorithm by white-box exploration.

To extract accurate models from the closed-source browsers, we employ *string-enhanced white-box exploration*. Our technique is similar in spirit to previous white-box exploration techniques used for automatic testing [17]–[19]. Unlike previous work, our technique builds a model from all the explored paths incrementally. Our technique also reasons directly about string operations rather than the individual byte-level operations that comprise those string operations, and we apply our technique to building models rather than generating test cases.

By reasoning directly about string operations, we can explore paths more efficiently, increasing the coverage achieved by the exploration per unit of time and improving the fidelity of our models. We expect directly reasoning about string operations will similarly improve the performance of other white-box exploration applications.

Preparation. A prerequisite for the exploration is to extract the prototype of the function that implements content sniffing and to identify the string functions used by that function.

2. Although a large portion of Safari is open-source as part of the WebKit project, Safari's content-sniffing algorithm is implemented in the *CFNetwork.dll* library, which is not part of the WebKit project.

For Internet Explorer 7, the online documentation at the Microsoft Developer Network (MSDN) states that content sniffing is implemented by the `FindMimeFromData` function [16]. MSDN also provides the prototype of `FindMimeFromData`, including the parameters and return values [20]. Using commercial off-the-self tools [21] as well as our own binary analysis tools [22], [23], we identified the string operations used by `FindMimeFromData` and the function that implements Safari 3.1's content-sniffing algorithm after some dynamic analysis and a few hours of manual reverse engineering.

Exploration. We build a model of the content-sniffing algorithm incrementally by iteratively generating inputs that traverse new execution paths in the program. In each iteration, we send an input to the program, which runs in a symbolic execution module that executes the program on both symbolic and concrete inputs. The symbolic execution module produces a *path predicate*, a conjunction of Boolean constraints on the input that captures how the execution path processes the input. From this path predicate, an input generator produces a new input by negating one of the constraints in the path predicate and solving the modified predicate. The input generator repeats this process for each constraint in the path predicate, generating many potential inputs for the next iteration. A path selector assigns priorities to these potential inputs and selects the input for the next iteration. We start the iterative exploration process with an initial input, called the seed, and continue exploring paths until there are no more paths to explore or until a user-specified maximum running time is exhausted. Once the exploration finishes, we output the disjunction of the path predicates as a model of the explored function.

String enhancements. String-enhanced white-box exploration improves white-box exploration by including string constraints in the path predicate. The input generator translates those string constraints into constraints understood by the constraint solver. We process strings in three steps:

1) Instead of generating constraints from the byte-level operations performed by string functions, the symbolic execution module generates constraints based on the output of these string functions using abstract string operators.
2) The input generator translates the abstract string operations into a language of arrays and integers understood by an off-the-shelf solver [24] by representing strings as a length variable and an array of some maximum length.
3) The input generator uses the output of the solver to build an input that starts a new iteration of the exploration.

These steps, as well as the abstract string operators, are detailed in [23].

By using string operators, we abstract the underlying string representation, letting us use the same framework for multiple languages. For example, we can apply our framework to the content-sniffing algorithm of Internet Explorer 7, which uses C strings (where strings are often represented as null-terminated character arrays), as well as to the content-sniffing algorithm of Safari 3.1, which uses a C++ string library (where strings are represented as objects containing a character array and an explicit length).

Even though no string constraint solver was publicly available during the course of this work, we designed our abstract string syntax so that it could use such a solver whenever available. Simultaneous work reports on solvers that support a theory of strings [25]–[27]. Thus, rather than translating the abstract string operations into a theory of arrays and integers, we could easily generate constraints in a theory of strings instead, benefiting from the performance improvements provided by these specialized solvers.

2.4. Content-Sniffing Algorithms

We analyze the content-sniffing algorithms used by four browsers: Internet Explorer 7, Firefox 3, Safari 3.1, and Google Chrome. We discover that the algorithms follow roughly the same design but that subtle differences between the algorithms have dramatic consequences for security. We compare the algorithms on several key points: the number of bytes used by the algorithm, the conditions that trigger sniffing, the signatures themselves, and restrictions on the HTML signature. We also discuss the "fast path" we observe in one browser.

Buffer size. We find that each browser limits content sniffing to the initial bytes of each HTTP response but that the number of bytes they consider varies by browser. Internet Explorer 7 uses 256 bytes. Firefox 3 and Safari 3.1 use 1024 bytes. Google Chrome uses 512 bytes, which matches the draft HTML 5 specification [5]. To be conservative, a server should filter uploaded content based on the maximum buffer size used by browsers: 1024 bytes.

Trigger conditions. We find that some HTTP responses trigger content sniffing but that others do not. Browsers determine whether to sniff based on the `Content-Type` header, but the specific values that trigger content sniffing vary widely. All four browsers sniff when the response lacks a `Content-Type` header. Beyond this behaviour, there is little commonality. Internet Explorer 7 sniffs if the header contains one of 35 "known" values listed in Table 4 in the Appendix (of which only 26 are documented in MSDN [16]). Firefox sniffs if the header contains a "bogus" value such as `*/*` or an invalid value that lacks a slash. Google Chrome triggers its content-sniffing algorithm with these bogus values as well as `application/unknown` and `unknown/unknown`.

image/jpeg	Signature
IE 7	DATA[0:1] == 0xffd8
Firefox 3	DATA[0:2] == 0xffd8ff
Safari 3.1	DATA[0:3] == 0xffd8ffe0
Chrome	DATA[0:2] == 0xffd8ff
image/gif	**Signature**
IE 7	(strncasecmp(DATA,"GIF87",5) == 0) || (strncasecmp(DATA,"GIF89",5) == 0)
Firefox 3	strncmp(DATA,"GIF8",4) == 0
Safari 3.1	N/A
Chrome	(strncmp(DATA,"GIF87a",6) == 0) || (strncmp(DATA,"GIF89a",6) == 0)
image/png	**Signature**
IE 7	(DATA[0:3] == 0x89504e47) && (DATA[4:7] == 0x0d0a1a0a)
Firefox 3	DATA[0:3] == 0x89504e47
Safari 3.1	N/A
Chrome	(DATA[0:3] == 0x89504e47) && (DATA[4:7] == 0x0d0a1a0a)
image/bmp	**Signature**
IE 7	(DATA[0:1] == 0x424d) && (DATA[6:9] == 0x00000000)
Firefox 3	DATA[0:1] == 0x424d
Safari 3.1	N/A
Chrome	DATA[0:1] == 0x424d

Table 1. Signatures for four popular image formats. `DATA` is the sniffing buffer. The nomenclature is detailed in the Appendix.

Signatures. We find that each browser employs different signatures. Table 1 shows the different signatures for four popular image types. Understanding the exact signatures used by browsers, especially the HTML signature, is crucial in constructing content-sniffing XSS attacks. The HTML signatures used by browsers differ not only in the set of HTML tags, but also in how the algorithm searches for those tags. Internet Explorer 7 and Safari 3.1 use permissive HTML signatures that search the full sniffing buffer (256 bytes and 1024 bytes, respectively) for predefined HTML tags. Firefox 3 and Google Chrome, however, use strict HTML signatures that require the first non-whitespace character to begin one of the predefined tags. The permissive HTML signatures in Internet Explorer 7 and Safari 3.1 let attackers construct chameleon documents because a file that begins `GIF89a<html>` matches both the GIF and the HTML signature. Table 2 presents the union of the HTML signatures used by the four browsers. These browsers will not treat a file as HTML if it does not match this signature.

Restrictions. We find that some browsers restrict when certain MIME types can be sniffed. For example, Google Chrome restricts which `Content-Type` headers can be sniffed as HTML to avoid privilege escalation (see Section 3). Table 5 in the Appendix shows which `Content-Type` header values each browser is willing to sniff as HTML.

text/html Signature
`(strncmp(PTR,"<!",2) == 0) ||`
`(strncmp(PTR,"<?",2) == 0) ||`
`(strcasestr(DATA,"<HTML") != 0) ||`
`(strcasestr(DATA,"<SCRIPT") != 0) ||`
`(strcasestr(DATA,"<TITLE") != 0) ||`
`(strcasestr(DATA,"<BODY") != 0) ||`
`(strcasestr(DATA,"<HEAD") != 0) ||`
`(strcasestr(DATA,"<PLAINTEXT") != 0) ||`
`(strcasestr(DATA,"<TABLE") != 0) ||`
`(strcasestr(DATA,"<IMG") != 0) ||`
`(strcasestr(DATA,"<PRE") != 0) ||`
`(strcasestr(DATA,"text/html") != 0) ||`
`(strcasestr(DATA,"<A") != 0) ||`
`(strncasecmp(PTR,"<FRAMESET",9) == 0) ||`
`(strncasecmp(PTR,"<IFRAME",7) == 0) ||`
`(strncasecmp(PTR,"<LINK",5) == 0) ||`
`(strncasecmp(PTR,"<BASE",5) == 0) ||`
`(strncasecmp(PTR,"<STYLE",6) == 0) ||`
`(strncasecmp(PTR,"<DIV",4) == 0) ||`
`(strncasecmp(PTR,"<P",2) == 0) ||`
`(strncasecmp(PTR,"<FONT",5) == 0) ||`
`(strncasecmp(PTR,"<APPLET",7) == 0) ||`
`(strncasecmp(PTR,"<META",5) == 0) ||`
`(strncasecmp(PTR,"<CENTER",7) == 0) ||`
`(strncasecmp(PTR,"<FORM",5) == 0) ||`
`(strncasecmp(PTR,"<ISINDEX",8) == 0) ||`
`(strncasecmp(PTR,"<H1",3) == 0) ||`
`(strncasecmp(PTR,"<H2",3) == 0) ||`
`(strncasecmp(PTR,"<H3",3) == 0) ||`
`(strncasecmp(PTR,"<H4",3) == 0) ||`
`(strncasecmp(PTR,"<H5",3) == 0) ||`
`(strncasecmp(PTR,"<H6",3) == 0) ||`
`(strncasecmp(PTR,"<B",2) == 0) ||`
`(strncasecmp(PTR,"<BR",3) == 0)`

Table 2. Union of HTML signatures. `PTR` is a pointer to the first non-whitespace byte of `DATA`.

Fast path. We find that, unlike other browsers, Internet Explorer 7 varies the order in which it applies its signatures according to the `Content-Type` header. If the header is `text/html`, `image/gif`, `image/jpeg`, `image/pjpeg`, `image/png`, `image/x-png`, or `application/pdf` and the content matches the signature for the indicated MIME type, then the algorithm skips the remaining signatures. Otherwise, the algorithm checks the signatures in the usual order.

Over time, Microsoft has added MIME types to this *fast path*. For example, in April 2008, Microsoft added `application/pdf` to the fast path to improve compatibility [28]. Microsoft classified this change as non-security related [29], but adding MIME types to the fast path makes construction of chameleon documents more difficult. If the chameleon matches a fast-path signature, the browser will not treat the chameleon as HTML. However, if the site's upload filter is more permissive than the browser's signature, the attacker can craft an exploit as we show in Section 2.5.

2.5. Concrete Attacks

In this section, we present two content-sniffing XSS attacks that we find by comparing our models of browser content-sniffing algorithms with the upload filters of two popular Web applications: HotCRP and Wikipedia. We implement and confirm the attacks using local installations of these sites.

HotCRP. HotCRP is the conference management Web application used by the 2009 IEEE Security & Privacy Symposium. HotCRP lets authors upload their papers in PDF or PostScript format.[3] Before accepting an upload, HotCRP checks whether the file appears to be in the specified format. For PDFs, HotCRP checks that the first bytes of the file are `%PDF-` (case insensitive), and for PostScript, HotCRP checks that the first bytes of the file are `%!PS-` (case insensitive).

HotCRP is vulnerable to a content-sniffing XSS attack because HotCRP will accept the chameleon document in Figure 1 as PostScript but Internet Explorer 7 will treat the same document as HTML. To mount the attack, the attacker submits a chameleon paper to the conference. When a reviewer attempts to view the paper, the browser treats the paper as HTML and runs the attacker's JavaScript as if the JavaScript were part of HotCRP, which lets the attacker give the paper a high score and recommend the paper for acceptance.

Wikipedia. Wikipedia is a popular Web site that lets users upload content in several formats, including SVG, PNG, GIF, JPEG, and Ogg/Theora [30]. The Wikipedia developers are aware of content-sniffing XSS attacks and have taken measures to protect their site. Before storing an uploaded file in its database, Wikipedia performs three checks:

1) Wikipedia checks whether the file matches one of the whitelisted MIME types. For example, Wikipedia's GIF signature checks if the file begins with `GIF`. Wikipedia uses PHP's MIME detection functions, which in turn use the signature database from the Unix `file` tool [31].
2) Wikipedia checks the first 1024 bytes for a set of blacklisted HTML tags, aiming to prevent browsers from treating the file as HTML.
3) Wikipedia uses several regular expressions to check that the file does not contain JavaScript.

Even though Wikipedia filters uploaded content, our analysis uncovers a subtle content-sniffing XSS attack. We construct the attack in three steps, each of which defeats one of the steps in Wikipedia's upload filter:

1) By beginning the file with `GIF88`, the attacker satisfies Wikipedia's requirement that the file begin with `GIF` without matching Internet Explorer 7's GIF signature, which requires that file begin with either `GIF87` or `GIF89`.
2) Wikipedia's blacklist of HTML tags is incomplete and contains only 8 of the 33 tags needed. To circumvent the blacklist, the attacker includes the string `<a href`, which is not on Wikipedia's blacklist but causes the file to match Internet Explorer 7's HTML signature.
3) To evade Wikipedia's regular expressions, the attacker can include JavaScript as follows:

```
<object src="about:blank"
  onerror="... JavaScript ...">
</object>
```

Although the fast path usually protects GIF images in Internet Explorer 7, a file constructed in this way passes Wikipedia's upload filter but is treated as HTML by Internet Explorer 7. To complete the cross-site scripting attack, the attacker uploads this file to Wikipedia and directs the user to view the file.

Wikipedia's PNG signature can be exploited using a similar attack because the signature contains only the first four of the eight bytes in Internet Explorer 7's PNG signature. Variants on this attack also affect other Web sites that use PHP's built-in MIME detection functions and the Unix `file` tool. These attacks demonstrate the importance of extracting precise models because the attacks hinge on subtle differences between the upload filter used by Wikipedia and the content-sniffing algorithm used by the browser.

The production instance of Wikipedia mitigates content-sniffing XSS attacks by hosting uploaded content on a separate domain. This approach does limit the severity of this vulnerability, but the installable version of Wikipedia, `mediawiki`, which is used by over 750 Web sites in the English language alone [32], hosts uploaded user content on-domain in the default configuration and is fully vulnerable to content-sniffing XSS attacks. After we reported this vulnerability to Wikipedia, Wikipedia has improved its upload filter to prevent these attacks.

3. Defenses

In this section, we describe two defenses against content-sniffing XSS attacks. First, we use our models to construct a secure upload filter that protects sites against content-sniffing XSS attacks. Second, we propose addressing the root cause of content-sniffing XSS attacks by securing the browser's content-sniffing algorithm.

Secure filtering. Based on the models we extract from the browsers, we implement an upload filter in 75 lines of Perl that protects Web sites from content-sniffing XSS attacks. Our filter uses the union HTML signature in Table 2. If a file passes the filter, the content is guaranteed not to be interpreted as HTML by Internet Explorer 7, Firefox 3,

3. A conference organizer can disable either paper format.

Safari 3.1, and Google Chrome. Using our filter, Web sites can block potentially malicious user-uploaded content that those browsers might treat as HTML.

Securing Sniffing. The secure filtering defense requires each Web site and proxy to adopt our filter. In parallel with this effort, browser vendors can mitigate content-sniffing XSS attacks against legacy Web sites by improving their content-sniffing algorithms. In the remainder of this section, we formulate a threat model for content-sniffing XSS attacks and propose two principles for designing a secure content-sniffing algorithm. We analyze the security and compatibility properties of an algorithm based on these principles.

3.1. Threat Model

We define a precise threat model for reasoning about content-sniffing XSS attacks. There are three principals in our threat model: the *attacker*, the *user* and the *honest Web site*. In a typical attack, the attacker uploads malicious content to the honest Web site and then directs the user's browser to render that content. We base our threat model on the standard Web attacker threat model [33]. Even though the Web attacker has more abilities than are strictly necessary to carry out a content-sniffing XSS attack, we use this threat model to ensure our defenses are robust.

- **Attacker abilities.** The attacker owns and operates a Web site with an untrusted domain name, canonically https://attacker.com/. These abilities can all be purchased on the open market for a nominal cost.
- **User behavior.** The user visits https://attacker.com/, but does not treat attacker.com as if it were a trusted site. For example, the user does not enter any passwords at attacker.com. When the user visits attacker.com, the attacker is "introduced" to the user's browser, letting the attacker redirect the user to arbitrary URLs. This assumption captures a central principle of Web security: browsers ought to protect users from malicious sites.
- **Honest Web site behavior.** The honest Web site lets the attacker upload content and then makes that content available at some URL. For example, a social networking site might let its users (who are potential attackers) upload images or videos. We assume that the honest site restricts what content the attacker can upload.

The most challenging part of constructing a useful threat model is characterizing how honest Web sites restrict uploads. For example, some honest sites (e.g., file storage services) might let users upload arbitrary content, whereas other sites might restrict the type of uploaded content (e.g., photograph sharing services) and perform different amounts of validation before serving the content to other users. Based on our case studies, we believe that many sites either restrict the `Content-Type`s they serve or filter content when uploaded (or both):

- **Restrict Content-Type.** Some Web sites restrict the `Content-Type` header they use when serving content uploaded by users. For example, a social networking Web site might enforce that its servers attach a `Content-Type` header beginning with `image/` to photographs, or a conference management Web application might serve papers only with a `Content-Type` header of `application/pdf` or `application/postscript`.
- **Filter uploads.** When users upload content, some sites use a function like PHP's `finfo_file` to check the initial bytes of the file to verify that the content conforms to the appropriate MIME type. For example, a photo sharing site might verify that uploaded files actually appear to be images and a conference management Web site might check that uploaded documents actually appear to be in PDF or PostScript format. Although not all MIME types can be recognized by their initial bytes, we assume sites only accept types commonly used on the Web. For these types, the initial bytes are dispositive.

We also assume that the honest site uses standard XSS defenses [34] to sanitize untrusted portions of HTML documents. However, we assume the honest site does not apply these sanitizers to non-HTML content because using an HTML sanitizer, such as PHP's `htmlentities`, on an image makes little sense because converting < characters to `<` would cause the image to render incorrectly.

Attacker goal. The attacker's goal is to mount an XSS attack against the honest site. More precisely, the attacker's goal is to run a malicious script in the honest site's security origin in the user's browser. In particular, we focus on attacks that leverage content sniffing to evade standard XSS defenses.

3.2. Design Principles

Content-sniffing algorithms trade off security and compatibility. To guide our design of a more secure content-sniffing algorithm, we propose two principles that help the algorithm maximize compatibility and achieve security.

- **Avoid privilege escalation.** Browsers assign different privileges to different MIME types. A content-sniffing algorithm avoids *privilege escalation* if the algorithm refuses to upgrade one MIME type to another of higher privilege. For example, the algorithm should not upgrade a response with a valid `Content-Type` header to `text/html` because HTML has the highest privilege (i.e., HTML can run arbitrary script).
- **Use prefix-disjoint signatures.** A content-sniffing algorithm uses *prefix-disjoint signatures* if its HTML signature does not share a prefix with a signature for another type commonly used on the Web. More precisely, a set of signatures is prefix-disjoint if there

does not exist two distinct sequences of bytes with a common prefix such that one matches the HTML signature and the other matches a signature for a non-HTML type commonly used on the Web. Firefox 3 and Google Chrome adhere to this principle, but Internet Explorer 7 and Safari 3.1 do not.

3.3. Security Analysis

Avoiding privilege escalation protects Web sites that restrict the values of the `Content-Type` header they attach to untrusted content because the browser will not upgrade attacker-supplied content to HTML (or another dangerous type) and will not run the attacker's malicious JavaScript. Unfortunately, avoiding privilege escalation is insufficient to protect all sites that filter uploads. For example, if a site serves content without a `Content-Type` header (e.g., if the site stores uploaded files in the file system and the Web server does not recognize the file extension), then the browser might sniff the uploaded content as HTML, opening the site up to attack.

Prefix-disjoint signatures, however, protect Web sites that filter uploaded content even if those sites use signatures that differ from the ones used by the browsers. If the site's signature is more strict than the browser's signature, then files accepted by the server will be sniffed correctly by the browser. If the site's signature is less strict (i.e., uses fewer initial bytes), then the site will be protected from content-sniffing XSS attacks in a browser that uses prefix-disjoint signatures. For example, suppose that the site acts like Wikipedia and checks only the first 4 of the initial 8 byte sequence required by the PNG standard [35]. If the browser uses prefix-disjoint signatures, no extension of this 4-byte sequence will match the HTML signature because this sequence can be extended to match the PNG signature. Even if the rest of the document consists of HTML tags, a browser that employs prefix-disjoint signatures will not treat the file as HTML and will prevent the attacker from crafting an exploit like the one in Section 2.5.

The HTML signature used by Internet Explorer 7 and Safari 3.1 is not prefix-disjoint because the signature searches for known HTML tags ignoring the initial bytes of the content, which might contain a signature for another type. For example, the string `GIF87a<html>` matches both the GIF signature and the HTML signature. Firefox 3 and Google Chrome use a strict HTML signature that requires the first non-whitespace characters to be a known HTML tag. According to our experiments on the Google search database (see Section 3.4), tolerating leading white space matches 9% more documents than requiring the initial characters of the content-sniffing buffer to be a known HTML tag. We recommend this HTML signature because the signature is prefix-disjoint from the other signatures.

3.4. Compatibility Evaluation

To evaluate the compatibility of our principles for secure content sniffing, we implement a content-sniffing algorithm that follows both of our design principles and collaborate with Google to ship the algorithm in Google Chrome. We use the following process to design the algorithm:

1) We evaluate the compatibility of our design principles over Google's search database, which contains billions of Web documents.
2) Google's quality assurance team manually tests our implementation for compatibility with the 500 most popular Web sites.
3) We deploy the algorithm to millions of users and improve the algorithm using aggregate metrics.

Search database. To avoid privilege escalation, our content-sniffing algorithm does not sniff HTML from most `Content-Type` values. To evaluate whether this behavior is compatible with the Web, we run a map-reduce query [36] over Google's search database. One limitation of this approach is that each page in the database contributes equally to the statistics, but users visit some pages (such as the CNN home page) much more often than other pages. The other two steps in our evaluation attempt to correct for this bias. From this data, we make the following observations:

- `<!DOCTYPE html` is the most frequently occurring initial HTML tag in documents that lack a `Content-Type` header. (We assign these documents a relative frequency of 1.)
- `<html` is the next most frequently occurring initial HTML tag in documents missing a `Content-Type` header. This occurs with relative frequency 0.612. For clarity, we limit the remainder of our statistics to this tag, but the results are similar if we consider all valid HTML tags.
- `<html` occurs as the initial bytes of documents with a `Content-Type` of `text/plain` with relative frequency 0.556, which is approximately the same relative frequency as for documents with a `Content-Type` of `unknown/unknown`.
- `<html` occurs as the initial bytes of documents with a bogus `Content-Type` (i.e., missing a slash) with relative frequency 0.059.
- When the `Content-Type` is valid, HTML tags occur with relative frequency less than 0.001.

From these observations, we conclude that, with the possible exception of `text/plain`, a content-sniffing algorithm can avoid privilege escalation by limiting when it sniffs HTML and remain compatible with a large percentage of the Web. From these observations, we do not draw a conclusion about `text/plain` because the data indicates that not sniffing HTML from `text/plain` is roughly as compatible as not sniffing HTML from `unknown/unknown`,

Signature	Mime Type	Percentage
`DATA[0:2] == 0xffd8ff`	image/jpeg	58.50%
`strncmp(DATA,"GIF89a",6) == 0`	image/gif	13.43%
`(DATA[0:3] == 0x89504e47) && (DATA[4:7] == 0x0d0a1a0a)`	image/png	5.50%
`strncasecmp(PTR,"<SCRIPT",7) == 0`	text/html	16.11%
`strncasecmp(PTR,"<HTML",5) == 0`	text/html	1.25%
`strncmp(PTR,"<?xml",5) == 0`	application/xml	1.10%

Table 3. The most popular signatures according to statistics collected from opt-in Google Chrome users. `PTR` is a pointer to the first non-whitespace byte of `DATA`.

yet none of the other major browsers sniff HTML from `unknown/unknown`. In our implementation, we choose to sniff HTML from `unknown/unknown` but not from `text/plain` because `unknown/unknown` is not a valid MIME type.

Top 500 sites. We implement a content-sniffing algorithm for Google Chrome according to both of our design principles. To evaluate compatibility, the Google Chrome quality assurance team manually analyzed the 500 most popular Web sites both with and without our content-sniffing algorithm. With the algorithm disabled, the team found a number of incompatibilities with major Web sites including Digg and United Airlines. With the content-sniffing algorithm enabled, the team found one incompatibility due to the algorithm not sniffing `application/x-shockwave-flash` from `text/plain`. However, every major browser is incompatible with this page, suggesting that this incompatibility is likely be resolved by the Web site operator.

Metrics. To improve the security of our algorithm, we instrument Google Chrome to collect metrics about the effectiveness of each signature from users who opt in to sharing their anonymous statistics. Based on this data, we find that six signatures (see Table 3) are responsible for 96% of the time the content sniffing algorithm changes the MIME type of an HTTP response. Based on this data, we remove over half of the signatures used by the initial algorithm. This change has a negligible impact on compatibility because these signatures trigger less than 0.004% of the time the content sniffing algorithm is invoked. Removing these signatures reduces the attack surface presented by the algorithm. Google has deployed our modified algorithm to all users of Google Chrome.

3.5. Adoption

In addition to being deployed in Google Chrome, our design principles have been standardized by the HTML 5 working group and adopted in part by Internet Explorer 8.

Standardization. The HTML 5 working group has adopted both of our content-sniffing principles in the draft HTML 5 specification [5]. The current draft advocates using prefix-disjoint signatures and classifies MIME types as either *safe* or *scriptable*. Content served with a safe MIME type carries no origin, but content served with a scriptable MIME type conveys the (perhaps limited) authority of its origin. The specification lets browsers sniff safe types from HTTP responses with valid `Content-Type`s (such as `text/plain`) but forbids browsers from sniffing scriptable types from these responses, avoiding privilege escalation.

Internet Explorer 8. The content-sniffing algorithm in Internet Explorer 8 differs from the algorithm in Internet Explorer 7. The new algorithm does not sniff HTML from HTTP responses with a `Content-Type` header that begins with the bytes `image/` [11], partially avoiding privilege escalation. This change significantly reduces the content-sniffing XSS attack surface, but it does not mitigate attacks against sites, such as HotCRP, that accept non-image uploads from untrusted users.

4. Related Work

In this section, we relate the current approaches used by sites that allow user uploads. These approaches provide an incomplete defense against content-sniffing XSS attacks. We also describe historical instances of content-sniffing XSS and related attacks.

Transform content. Web sites can defend themselves against content-sniffing XSS attacks by transforming user uploads. For example, Flickr converts user-uploaded PNG images to JPEG format. This saves on storage costs and makes it more difficult to construct chameleon documents because HTML content inside the PNG is often destroyed by the transformation. Unfortunately, this approach does not guarantee security because an attacker might be able to craft a chameleon that survives the transformation. Also, sites might have difficulty transforming non-media content, like text documents.

Host content off-domain. Some sites host user-supplied content on an untrusted domain. For example, Wikipedia hosts English-language articles at en.wikipedia.org but hosts

uploaded images at upload.wikimedia.org. Content-sniffing XSS attacks compromise the http://upload.wikimedia.org origin but not the http://en.wikipedia.org origin, which contains the user's session cookie. This approach has a couple of disadvantages. First, hosting uploads off-domain complicates the installation of redistributable Web applications like `phpBB`, Bugzilla, or `mediawiki`. Also, hosting uploads off-domain limits interaction with these uploads. For example, sites can display off-domain images but cannot convert them to data URLs or use them in SVG filters. Although hosting user-uploaded content off-domain is not a complete defense, the approach provides defense-in-depth and reduces the site's attack surface.

Disable content sniffing. Users can disable content sniffing using advanced browser options, at the cost of compatibility. Sites can disable content sniffing for an individual HTTP response by adding a `Content-Disposition` header with the value `attachment` [37], but this causes the browser to download the file instead of rendering its contents. Another approach, used by Gmail, to disable content sniffing is to pad `text/plain` attachments with 256 leading whitespace characters to exhaust Internet Explorer's sniffing buffer.

Internet Explorer 8 lets sites disable content sniffing for an individual HTTP response (without triggering the download handler) by including an `X-Content-Type-Options` header with the value `nosniff` [38]. This feature lets sites opt out of content sniffing but requires sites to modify their behavior. We believe this header is complementary to securing the content-sniffing algorithm itself, which protects sites that do not upgrade.

Content-sniffing XSS attacks. Previous references to content-sniffing XSS attacks focus on the construction of chameleon documents that Internet Explorer sniffs as HTML. Four years ago, a blog post [2] discusses a JPEG/HTML chameleon. A 2006 full disclosure post [4] describes a content-sniffing XSS attack that exploits an incorrect `Content-Type` header. More recently, PNG and PDF chameleons have been used to launch content-sniffing XSS attacks [3], [12], [39], [40]. Spammers have reportedly used similar attacks to upload text files containing HTML to open wikis [3]. Many of the example exploits in these references no longer work, suggesting that Internet Explorer's content-sniffing algorithm has evolved over time by adding MIME types to the fast path.

JAR URI Scheme. Although not a content-sniffing vulnerability as such, Firefox 2.0.0.9 contains a vulnerability caused by treating one type of content as another. Firefox supports extracting HTML documents from ZIP archives using the `jar` URI scheme. If a site lets an attacker upload a ZIP archive, the attacker can instruct Firefox to unzip the archive and render the HTML inside [41]. Worse, because the ZIP parser is tolerant of malformed archives, an attacker can create chameleon ZIP archives that appear to be images. To resolve this issue, Firefox now requires the archives to be served with specific MIME types.

5. Conclusions

Browser content-sniffing algorithms have long been one of the least-understood facets of the browser security landscape. In this paper, we study content-sniffing XSS attacks and defenses. To understand content-sniffing XSS attacks, we use string-enhanced white-box exploration and source code inspection to construct high-fidelity models of the content-sniffing algorithms used by Internet Explorer 7, Firefox 3, Safari 3.1, and Google Chrome. We use these models to construct attacks against two Web applications: HotCRP and Wikipedia.

We describe two defenses for these attacks. For Web sites, we provide a filter based on our models that blocks content-sniffing XSS attacks. To protect sites that do not deploy our filter, we propose two design principles for securing browser content-sniffing algorithms: avoid privilege escalation and use prefix-disjoint signatures. We evaluate the security of these principles in a threat model based on case studies, and we evaluate the compatibility of these principles using Google's search database and metrics from over a billion of HTTP responses.

We implement a content-sniffing algorithm based on our principles and deploy the algorithm to real users in Google Chrome. Our principles have been incorporated into the draft HTML 5 specification and partially adopted by Internet Explorer 8. We look forward to continue working with browser vendors to converge their content sniffers towards a secure, standardized algorithm.

Acknowledgements

We would like to thank Stephen McCamant, Rhishikesh Limaye, Susmit Jha, and Sanjit A. Seshia who collaborated in the design of the abstract string syntax. We also thank Darin Adler, Darin Fisher, Ian Hickson, Collin Jackson, Eric Lawrence, and Boris Zbarsky for many helpful discussions on content sniffing. Finally, our thanks to Chris Karlof, Adrian Mettler, and the anonymous reviewers for their insightful comments on this document.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] "Firefox bug 175848," https://bugzilla.mozilla.org/show_bug.cgi?id=175848.

[2] "Getting around Internet Explorer MIME type mangling," http://weblog.philringnalda.com/2004/04/06/getting-around-ies-mime-type-mangling.

[3] "Internet Explorer facilitates XSS," http://www.splitbrain.org/blog/2007-02/12-internet_explorer_facilitates_cross_site_scripting.

[4] "SMF upload XSS vulnerability," http://seclists.org/fulldisclosure/2006/Dec/0079.html.

[5] I. Hickson *et al.*, "HTML 5 Working Draft," http://www.whatwg.org/specs/web-apps/current-work/.

[6] N. Freed and N. Borenstein, "RFC 2045: Multipurpose Internet Mail Extensions (MIME) part one: Format of Internet message bodies," Nov. 1996.

[7] ——, "RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types," Nov. 1996.

[8] K. Moore, "RFC 2047: Multipurpose Internet Mail Extensions (MIME) part three: Message header extensions for non-ASCII text," Nov. 1996.

[9] "Apache bug 13986," https://issues.apache.org/bugzilla/show_bug.cgi?id=13986.

[10] "EXIF.org," http://www.exif.org/.

[11] "Internet Explorer 8 security part V: Comprehensive protection," http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx.

[12] "Internet Explorer XSS exploit door," http://tweakers.net/nieuws/47643/xss-exploit-door-microsoft-betiteld-als-by-design.html.

[13] "Wikipedia," http://www.wikipedia.org.

[14] "HotCRP conference management software," http://www.cs.ucla.edu/~kohler/hotcrp/.

[15] "WineHQ," http://www.winehq.org/.

[16] "MSDN: MIME type detection in Internet Explorer," http://msdn.microsoft.com/en-us/library/ms775147.aspx.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, Virginia, October 2006.

[18] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[19] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Annual Network and Distributed System Security Symposium*, San Diego, California, February 2008.

[20] "MSDN: FindMimeFromData function," http://msdn.microsoft.com/en-us/library/ms775107(VS.85).aspx.

[21] "The IDA Pro disassembler and debugger," http://www.hex-rays.com/idapro/.

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Hyderabad, India, December 2008, Keynote invited paper.

[23] J. Caballero, S. McCamant, A. Barth, and D. Song, "Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-36, Mar 2009.

[24] V. Ganesh and D. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the Computer Aided Verification Conference*, Berlin, Germany, August 2007.

[25] N. Bjorner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, York, United Kingdom, March 2009.

[26] P. Hooimeijer and W. Weimer, "A decision procedure for subset constraints over regular languages," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[27] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A solver for string constraints," MIT CSAIL, Tech. Rep. MIT-CSAIL-TR-2009-004, Feb. 2009.

[28] "Microsoft KB945686," http://support.microsoft.com/kb/945686/.

[29] "Microsoft KB944533," http://support.microsoft.com/kb/944533.

[30] "Wikipedia image use policy," http://en.wikipedia.org/wiki/Image_use_policy.

[31] "Fine free file command," http://darwinsys.com/file/.

[32] "Sites using mediawiki/en," http://www.mediawiki.org/wiki/Sites_using_MediaWiki/en.

[33] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," in *Proceedings of the Usenix Security Symposium*, San Jose, California, July 2008.

[34] M. Martin and M. S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in *Proceedings of the USENIX Security Symposium*, San Jose, California, July 2008.

[35] "Portable Network Graphics specification, w3c/iso/iec version," http://www.libpng.org/pub/png/spec/iso/.

[36] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, December 2004.

[37] R. Troost, S. Dorner, and K. Moore, "RFC 2183: Communicating presentation information in Internet messages: The content-disposition header field," Aug. 1997.

[38] "Internet Explorer 8 security part V: Comprehensive protection," http://blogs.msdn.com/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx.

[39] "The hazards of MIME sniffing," http://adblockplus.org/blog/the-hazards-of-mime-sniffing.

[40] "The downside of uploads," http://www.malevolent.com/weblog/archive/2008/02/26/uploads-mime-sniffing/.

[41] "Mozilla foundation security advisory 2007-37," http://www.mozilla.org/security/announce/2007/mfsa2007-37.html.

Appendix

Nomenclature. We adopt the following nomenclature to represent signatures precisely. `DATA` is a pointer to a buffer containing the first n bytes of the content, where n is the size of the content-sniffing buffer size for the particular browser. `DATA[x:y]`, where $n > y \geq x \geq 0$, is the subsequence of `DATA` beginning at offset x and ending at offset y (both offsets inclusive). For example, Internet Explorer 7 uses the following signature for `image/jpeg`: `DATA[0:1] == 0xffd8`. To match this signature, an HTTP response must contain at least two bytes, the first byte of the response must be `0xff`, and the second byte must be `0xd8`. We also use four functions to express signatures: `strncmp` for case-sensitive comparison, `strncasecmp` for case-insensitive comparison, `strstr` for case-sensitive search, and `strcasestr` for case-insensitive search.

Additional data. Table 4 presents the list of 35 MIME types that Internet Explorer 7 considers as "known" and thus trigger the content-sniffing algorithm. In addition to those `text/plain` and `application/octet-stream` also trigger the content-sniffing algorithm in Internet Explorer 7.

Table 5 presents `Content-Type` values that the different browsers are willing to upgrade to `text/html` if the corresponding signature is matched. In the table, Missing means that the value is absent, Bogus means that the value lacks a slash, and Known means that the value is in Table 4.

Documented	Undocumented
application/base64	(null)
application/java	application/x-cdf
application/macbinhex40	application/x-netcdf
application/pdf	application/xml
application/postscript	image/png
application/x-compressed	image/x-art
application/x-gzip-compressed	text/scriptlet
application/x-msdownload	text/xml
application/x-zip-compressed	video/x-msvideo
audio/basic	
audio/wav	
audio/x-aiff	
image/bmp	
image/gif	
image/jpeg	
image/pjpeg	
image/tiff	
image/x-emf	
image/x-jg	
image/x-png	
image/x-wmf	
image/x-xbitmap	
text/html	
text/richtext	
video/avi	
video/mpeg	

Table 4. Mime types that trigger content sniffing in Internet Explorer 7. Mime types `text/plain` and `application/octet-stream` also trigger the content-sniffing algorithm.

Content-Type	Chrome	IE 7	FF 3	Safari 3.1
Missing	yes	yes	yes	yes
Bogus	yes	no	yes	no
Known	no	yes	no	no
`*/*`	yes	no	yes	no
`application/unknown`	yes	no	no	no
`unknown/unknown`	yes	no	no	no
`text/plain`	no	yes	no	`.html` extension
`application/octet-stream`	no	yes	no	yes

Table 5. `Content-Type` values that can be upgraded to `text/html`. Missing means the value is absent. Bogus means the value lacks a slash. Known means the value is in Table 4.