Backslash Powered Scanning: Hunting Unknown Vulnerability Classes

James Kettle - james.kettle@portswigger.net - @albinowax

Abstract

Existing web scanners search for server-side injection vulnerabilities by throwing a canned list of technology-specific payloads at a target and looking for signatures - almost like an anti-virus. In this document, I'll share the conception and development of an alternative approach, capable of finding and confirming both known and unknown classes of injection vulnerabilities. Evolved from classic manual techniques, this approach reaps many of the benefits of manual testing including casual WAF evasion, a tiny network footprint, and flexibility in the face of input filtering.

True to its heritage, this approach also manages to harness some pitfalls that will be all too familiar to experienced manual testers. I'll share some of the more entertaining findings and lessons learned from unleashing this prototype on a few thousand sites, and release a purpose-built stealthy-scanning toolkit. Finally, I'll show how it can be taken far beyond injection hunting, leaving you with numerous leads for future research.

Outline

- Introduction
- Three Failures of Scanners
    - Rare Technology
    - Variants and Filters
    - Buried vulnerabilities
- Alternative Approach to Scanning
- Suspicious Input Transformations
- Response Diffing Technique
    - Core logic
    - Types of Mutation
    - Recognising Response Differences
- Hunting Findings
    - Scanning Distributed System
    - Sample Results
        - MySQL Injection
        - Filtered Code Injection
        - Old RCE
        - Regex Injection
        - Escaping Flaws
        - Semantic False Positives
        - Web Application Firewall
        - SOLR JSON Injection
    - Lessons Learned
- Further Research
    - Iterable Input Detection
    - Cold-start Bruteforce Attacks
- Conclusion

Introduction

Outside marketing brochures, web application scanners are widely regarded as only being fit for identifying 'low-hanging fruit' - vulnerabilities that are obvious and easily found by just about anyone. This is often a fair judgement; in comparison with manual testers, automated scanners' reliance on canned technology-specific payloads and innate lack of adaptability means even the most advanced scanners can fail to identify vulnerabilities obvious to a human. In some cases it's unfair - scanners are increasingly good at detecting client-side issues like Cross-Site Scripting, even identifying DOM-based XSS using both static[1] and dynamic[2] analysis. As Black-box scanners lack insight into what's happending server-side, they typically have a harder time with detection of server-side injection vulnerabilites like SQL injection, Code Injection, and OS Command Injection.

In this paper, I'll break down the three core blind spots in scanners' detection of server-side injection vulnerabilities, then show that by implementing an approach to scanning evolved from classic manual techniques, I was able to develop a scanner capable of detecting research-grade vulnerabilities far above low-hanging fruit. In particular, I will show that this scanner could have found Server-Side Template Injection[3] (SSTI) vulnerabilities prior to the vulnerability class being discovered. This scanner has been released as an open source extension to Burp Suite.

Three Failures of Scanners

Blind Spot 1: Rare Technology

Security through obscurity works against scanners. As an illustration, I'll look at SSTI , a vulnerability that arises when an application unsafely embeds user input into a template. Depending on the template engine in use, it may be possible to exploit this to gain arbitrary code execution and complete contror of the server. In order for a scanner to detect this vulnerability, it needs to be hard coded with a payload for each template engine. If your application is using a popular template engine like FreeMarker or Jinja, that's fine. But how many of the following template engines does your scanner support?

> Amber, Apache Velocity, action4JAVA, ASP.NET (Microsoft), ASP.NET (Mono), AutoGen, Beard, Blade, Blitz, Casper, CheetahTemplate, Chip Template Engine, Chunk Templates, CL-EMB, CodeCharge Studio, ColdFusion, Cottle, csharptemplates, CTPP, dbPager, Dermis, Django, DTL::Fast (port of Django templates), Djolt-objc, Dwoo, Dylan Server Pages, ECT, eRuby, FigDice, FreeMarker, Genshi (templating language), Go templates, Google-ctemplate, Grantlee Template System, GvTags, H2o, HAH, Haml, Hamlets, Handlebars, Hyperkit PHP/XML Template Engine, Histone template Engine, HTML-TEMPLATE, HTTL, Jade, JavaServer Pages, jin-template, Jinja, Jinja2, JScore, Kalahari, Kid (templating language), Liquid, Lofn, Lucee, Mako, Mars-Templater, MiniTemplator, mTemplate, Mustache, nTPL, Open Power Template, Obyx, Pebble, Outline, pHAML, PHP, PURE Unobtrusive Rendering Engine, pyratemp, QueryTemplates, RainTPL, Razor, Rythm, Scalate, Scurvy, Simphple, Smarty, StampTE, StringTemplate, SUIT Framework, Template Attribute Language, Twital, Template Blocks, Template Toolkit, Thymeleaf, TinyButStrong, Tonic, Toupl, Twig, Twirl, uBook Template, vlibTemplate, WebMacro, ZeniTPL, BabaJS, Rage, PlannerFw, Fenom

This list only includes the template engines well known enough to be recorded on Wikipedia. Michael Stepankin recently found a remote code execution vulnerability in Paypal[4] stemming from SSTI in Dust.js[5], a templating engine by LinkedIn conspiciously missing from the above list. This issue applies equally to anyone using the myriad obscure database languages out there, not to mention frameworks that distort code injection beyond comprehension. Scanners' forced assumptions about the backend technology stack can create bizarre side effects - running a webapp under SELinux may mean many scanners fail to detect Local File Include and External Entity Include vulnerabilities, since these are typically detected by reading the contents of /etc/passwd, an action SELinux may block[6].

If this wasn't the case, scanner vendors would be regularly releasing juicy vulnerabilities like SSTI, rather than them going unnoticed for years. Applications with obscure vulnerabilities are absolutely being scanned - during the early stages of my SSTI research when the issue was unpublished, a client of ours informed us that Burp Suite was reporting a false-positive XSS vulnerability on their site. When I investigated the site myself it quickly became apparent the 'false positive' was caused by a significantly more serious SSTI vulnerability. Ultimately, scanners have seriously degraded performance on applications using the long tail of obscure technologies.

Blind Spot 2: Variants and Filters

Consider a classic vulnerability in a well known language: blind code injection in PHP, inside a double-quoted string. A scanner can easily detect this by sending a payload to incude a time-delay:

```
".sleep(10)."
```

So far so good. But if the application happens to filter out parenthesis, we'll get a false negative although the application could still be exploited using

```
".`sleep 10`."
```

If there's a Web Application Firewall (WAF) looking for payloads containing the word 'sleep', we'll almost certainly get a false negative again. If the application is normalising input, we can probably still exploit it by using the Cyrillic e character in the hope that it gets normalised into e

```
".sl%D0%B5ep(10)."
```

And if the application is filtering "? Once again, we'll get a false negative, when the application is still easily exploitable:

```
{${sleep(10)}}
```

Of these three examples, I've encountered two personally during pentests and seen the third in a writeup by someone else. The design of scanners makes them easily thwarted by unexpected filters and variations. Scanners could of course send the payloads shown above, but those only cover three of numerous possible variations of a single vulnerability. Sending sufficient payloads to cover every variation of every vulnerability is fundamentally implausible at today's network speeds - I'll call this the Million Payload Problem. This means scanners are reduced to sending 'best-effort' payloads, and means even something as basic as using double quotes instead of single quotes to encapsulate SQL statements can annihilate a scanner's detection capabilities.

Blind Spot 3: Buried Vulnerabilities

The following HTTP request is to an endpoint on Ebay that used to be vulnerable to arbitrary code execution via PHP injection. Where should a scanner try injecting its payloads?

```
GET /search/?q=david  HTTP/1.1
Host: sea.ebay.com.sg
User-Agent: Mozilla/5.0 etc Firefox/49.0
Accept: text/html
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://sea.ebay.com.sg/
Cookie: session=pZGFjciI6IjAkLCJlx2V4cCI6MTA4
Connection: close
Origin: null
X-Forwarded-For: 127.0.0.1
X-Forwarded-Host: evil.com
```

The obvious place to inject is the 'q' parameter, but that doesn't work. Neither does the Referer, User-Agent, or session cookie. An experienced pentester might try injecting in some headers that aren't present, like Origin, X-Forwarded-For, or X-Forwarded Host[7]. In this case, none of these would work either. By the time a scanner reaches this point, it's sent an awful lot of payloads without success. David Vieira-Kurz found it was possible to exploit this endpoint by passing a second q parameter, creating a malicious array server-side[8]:

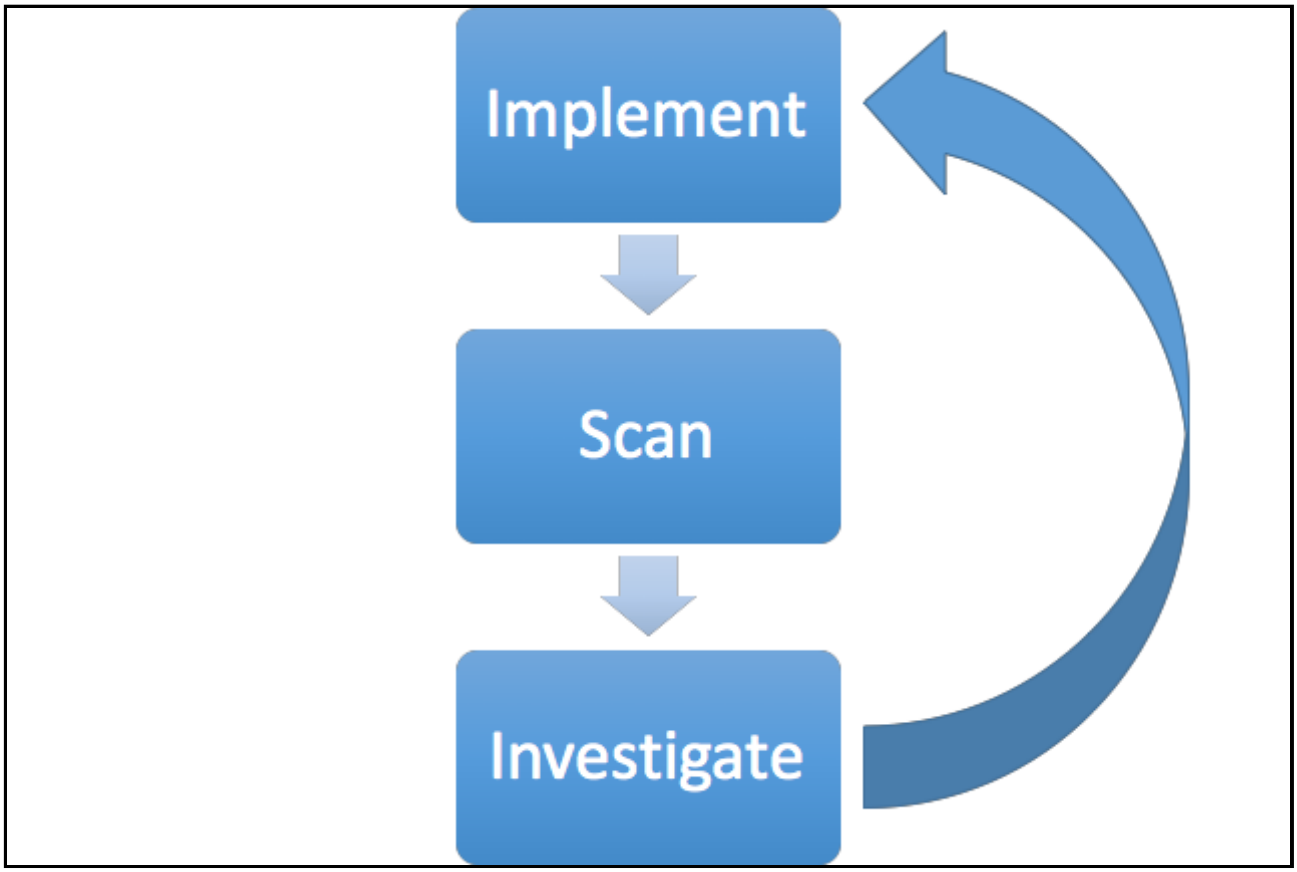
```
GET /search/?q=david&q[1]=sec{${phpinfo()}}
```

He tried this attack because the q parameter causes a search that has a spellchecker and also filters out certain keywords, which provided a clue that something interesting was happening server-side. Here we once again have a vulnerability that a scanner could detect only if it had no constraints on the number of payloads it could send to each endpoint. This example is an extreme case, but vulnerabilities in other rarely-useful inputs like the Accept-Language header are also likely to be missed.

An Alternative Approach to Scanning

At this point you know how to make an application more or less scanner-proof; just code it with an obscure web language, store data with a niche NoSQL variant with non-standard syntax, and layer a couple of WAFs on top for good measure. How is it that manual testers avoid these blind spots? The fundamental is their concept boring inputs, and interesting, suspicious or promsing inputs. David Vieira-Kurz's observation that an input had a spellchecker directly lead to him subjecting it to extensive auditing that would be a waste of time on your typical input.

We can learn from this. Rather than scanning for vulnerabilities, we need to scan for *interesting behaviour*. Then, having identified the tiny fraction of inputs that yield interesting behaviour, we can investigate further. This iterative approach to identifying vulnerabilities is both extremely flexible in what it can identify, and highly efficient. An input that doesn't yield any interesting results can be quickly discounted, saving time for sustained investigation of inputs that look more promising. The development of a scanner that uses this technique can also be approached in successive stages, as expressed in the following positive feedback cycle:



Suspicious Input Transformation Technique

The initial probe used to identify suspicious behaviour should be as simple and generic as possible. Take the following payload which exploits FreeMarker SSTI:

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("id")
}
```

We can easily roll this back to a more generic payload that will identify most template engines using a popular statement syntax:

```
${7*7} (expect 49)
```

Can we expand the coverage of this to detect generic code evaluation? We could try something like:

```
7*7 (expect 49)
```

but that will only work on numeric inputs. To detect injection into strings, we need something like:

```
\x41 (expect A)
```

However many languages, notably including SQL, don't support hex escapes. This probe can be made one step more generic, to support almost every language:

```
\\ (expect \)
```

At this point we have our very first probe for detecting suspicious input transformations. We can now move to the 'scan' stage, trying out this payload on a range of applications and seeing what it throws up. Provided the probe is good and the testbed is large enough (more on that later), we'll get a suitably sized set of results which we can manually investigate to find out what's interesting. In this case, the first step to understanding the behaviour was to look for other input transformations like \x41=>A. By comparing the application's handling of a known-bad escape sequence with other characters, we can gain subtle clues to which characters has special significance server-side. For example, using the baseline of \zz we can easily spot the anomaly:

```
\zz => \zz

\" => \"
\$ => \$
\{ => {
\x41 => \x41
```

This tells us that the { character has special significance. Having repeated and refined this manual investigation process a few times, we can loop back around to the 'Implement' stage and automate it. Here's a screenshot of the scanner's output on a page that is vulnerable to Markdown injection:

Suspicious Input Transformation

Issue:	**Suspicious Input Transformation**
Severity:	**High**
Confidence:	**Tentative**
Host:	**http://codepen.io**
Path:	**/preprocessors**

Note: This issue was generated by the Burp extension: protoScan2.

Issue detail

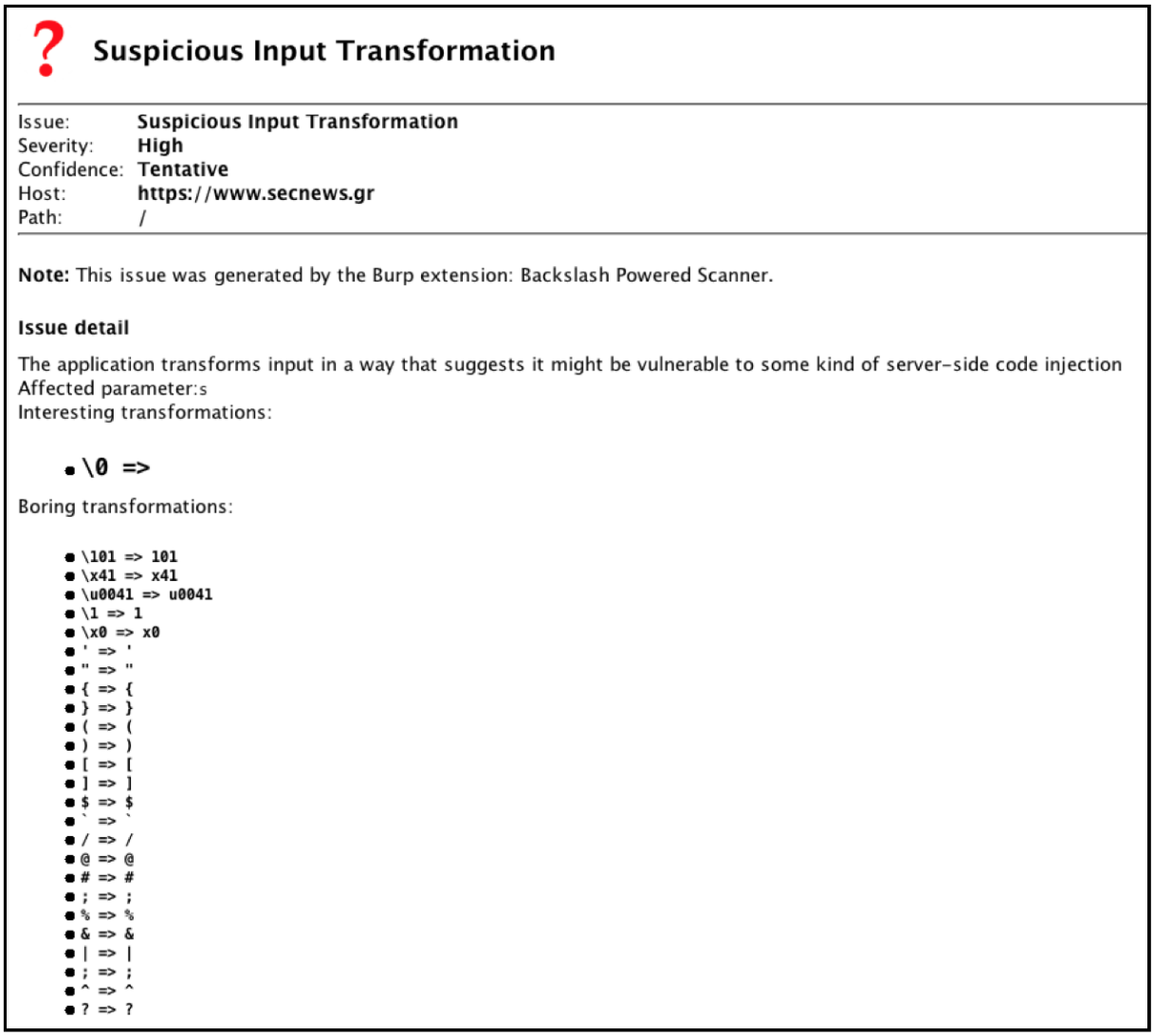
The application transforms input in a way that suggests it might be vulnerable to some kind of server-side code injection
Affected parameter:1
Interesting transformations:

- \{ => {
- { => {
- \} => }
- } => }
- \(=> (
- (=> (
- \) =>)
-) =>)
- \[=> [
- [=> [
- \] =>]
-] =>]
- \` => `
- ` => `
- \# => #
- # => #
- \& => &
- & => &
- \| => |
- | => |
- \^ => ^
- ^ => ^

Boring transformations:

- \101 => \101
- \x41 => \x41
- \u0041 => \u0041
- \0 => \0
- \1 => \1
- \' => \'
- \" => \"
- \$ => \$
- \/ => \/

And a page that isn't vulnerable to anything, but merely calls stripslashes() on the input:



Suspicious Input Transformation

Issue: **Suspicious Input Transformation**
Severity: **High**
Confidence: **Tentative**
Host: **https://www.secnews.gr**
Path: /

Note: This issue was generated by the Burp extension: Backslash Powered Scanner.

Issue detail

The application transforms input in a way that suggests it might be vulnerable to some kind of server-side code injection
Affected parameter:s
Interesting transformations:

- \0 =>

Boring transformations:

- \101 => 101
- \x41 => x41
- \u0041 => u0041
- \1 => 1
- \x0 => x0
- ' => '
- " => "
- { => {
- } => }
- (=> (
-) =>)
- [=> [
-] =>]
- $ => $
- ` => `
- / => /
- @ => @
- # => #
- ; => ;
- % => %
- & => &
- | => |
- ; => ;
- ^ => ^
- ? => ?

If you're aware of (or able to construct) targets that are definitely vulnerable you can verify the scanner's susceptability to false negatives. I found the scanner failed to identify vulnerabilities in JSON responses, since although the server would decode \\ to \, it would then escape the \ back to \\ when embedding it in a JSON string. This was easily fixed by JSON decoding responses where appropriate.

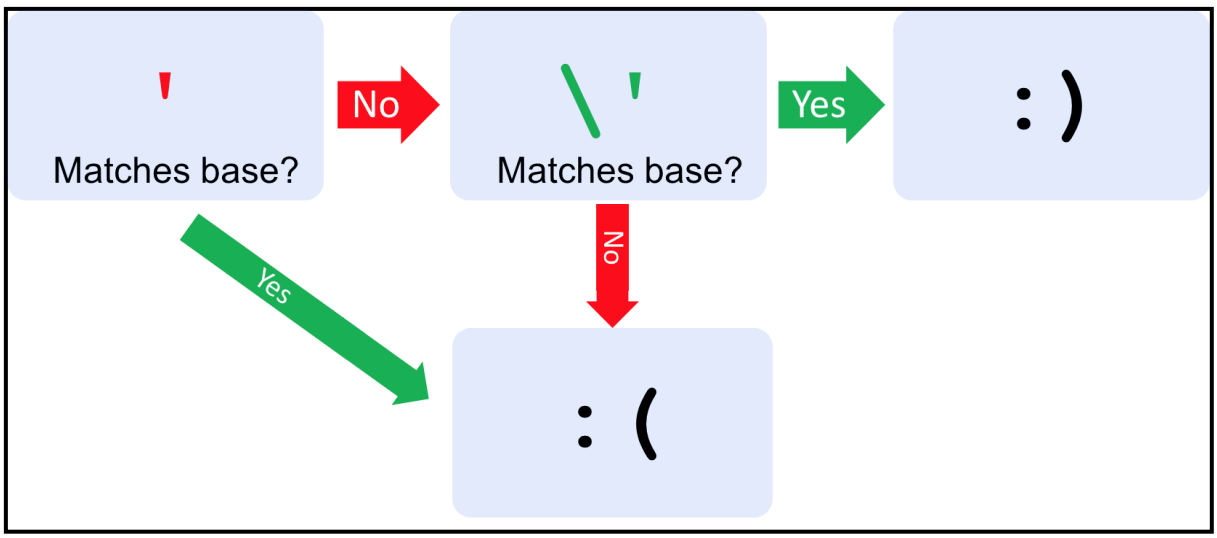
A more serious weakness with is that this approach relies on user input being reflected after it's been processed. If an application places user input in a SQL SELECT statement, but never displays that query, the vulnerability will be missed entirely. This is a fundamental flaw with relying on suspicious input transformations to detect vulnerabilities.

Guided Fuzzing Approach

Core Logic

We can avoid relying on input reflection by analysing the entire response and inferring whether our input caused a significant change. This is quite similar to a classic fuzzer (throw input at the application and see if it crashes), and something many pentesters will be familiar with partially automating using Burp Intruder and fuzzlists. We aren't limited to naively looking at status codes and grepping for error messages - using automation, we can recognise changes as subtle as a single word or empty line disappearing.

Just like a manual tester, we can gather further information using pairs of probes. First, we identify the normal response of the application by sending a probe containing random alphanumeric characters. This will be reffered to as the 'base' response'. If a probe containing ' consistently gets a response that's different from the base, we can infer that the ' character has a special significance to the application. This may not indicate a vulnerability - the application might just be rejecting inputs with '. Once again, we can use backslashes to escape our predicament. If the application responds to probes containing \' in the same way as random alphanumeric probes, we can infer that the anomalous response to ' is a caused by a failure to escape the character. This might make more sense in a diagram:

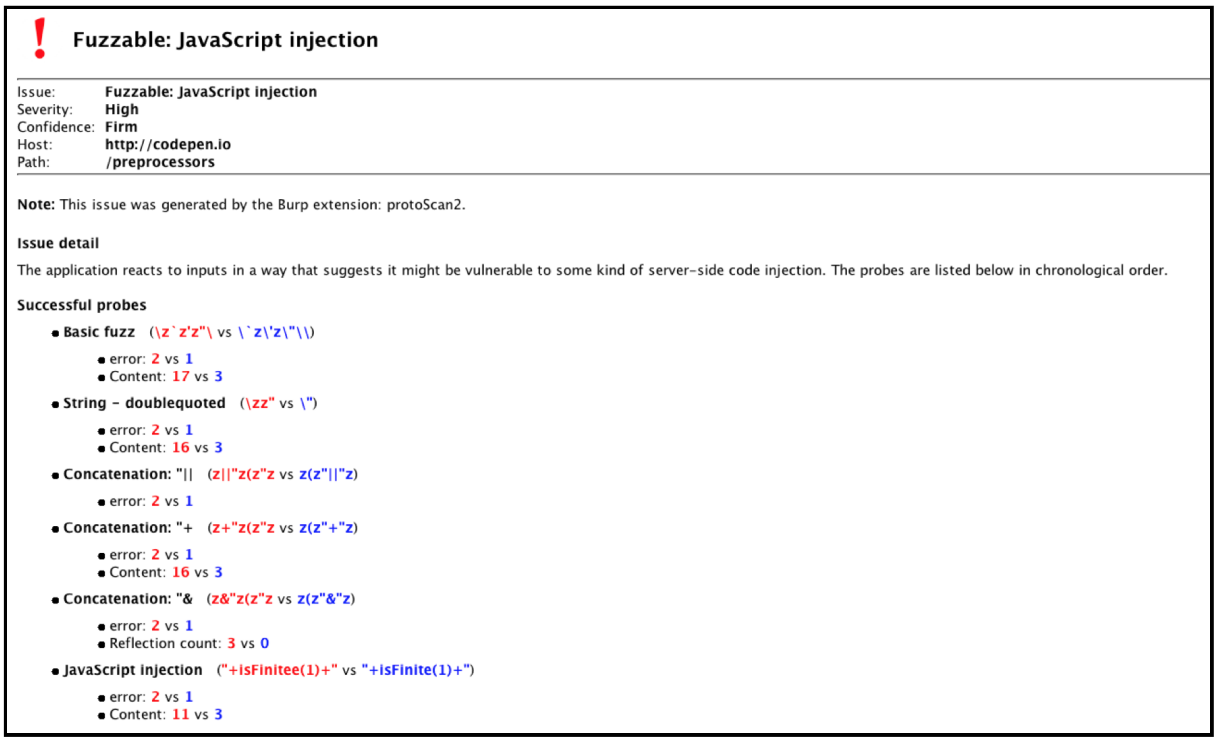


This technique isn't limited to identifying injection into strings. We can identify injections into various other contexts by using alternative probe-pairs. Each additional probe pair only requires a few lines of code, so we're already using quite a few:

```
' vs \' // single-quoted string
" vs \" // double-quoted string
7/0 vs 7/1 // number
${{ vs $}} // interpolation
/**/ vs /**/ // raw code
,99 vs ,1 // order-by
sprintz vs sprintf // function name
```

We can also string sequences of probe-pairs together, to iteratively gather more information on a potential vulnerability. When faced with injection into a string, Backslash Powered Scanner will first identify the type of quote in use, then the concatenation sequence, then identify whether function calls are possible, and finally try a list of language-specific functions to try and identify the backend language. The following screenshot shows the scanner's output when pointed at an application vulnerable to Server-Side JavaScript Injection.

Note that the infromation obtained in each stage is used by the following stage.



Fuzzable: JavaScript injection

Issue: **Fuzzable: JavaScript injection**
Severity: **High**
Confidence: **Firm**
Host: **http://codepen.io**
Path: **/preprocessors**

Note: This issue was generated by the Burp extension: protoScan2.

Issue detail

The application reacts to inputs in a way that suggests it might be vulnerable to some kind of server-side code injection. The probes are listed below in chronological order.

Successful probes

- **Basic fuzz** (\z`z'z"\ vs \`z\'z\"\\)
  - error: 2 vs 1
  - Content: 17 vs 3
- **String - doublequoted** (\zz" vs \")
  - error: 2 vs 1
  - Content: 16 vs 3
- **Concatenation: "||** (z||"z(z"z vs z(z"||"z)
  - error: 2 vs 1
- **Concatenation: "+** (z+"z(z"z vs z(z"+"z)
  - error: 2 vs 1
  - Content: 16 vs 3
- **Concatenation: "&** (z&"z(z"z vs z(z"&"z)
  - error: 2 vs 1
  - Reflection count: 3 vs 0
- **JavaScript injection** ("+isFinitee(1)+" vs "+isFinite(1)+")
  - error: 2 vs 1
  - Content: 11 vs 3

Types of Mutation

Applications handle modified inputs in one of two distinct ways. Some inputs, typically those where the input originates from a free-form text field like a comment, only display a distinct response when you trigger a syntax error server-side:

```
/post_comment?text=baseComment      200 OK
/post_comment?text=randomtext       200 OK
/post_comment?text=random'text      500 Oops
/post_comment?text=random\'text     200 OK
```

On other inputs, any deviation from the expected input triggers an error:

```
/profile?user=bob                   200 OK
/profile?user=randomtext            500 Oops
/profile?user=random'text           500 Oops
/profile?user=random\'text          500 Oops
/profile?user=bo'||'b               200 OK
/profile?user=bo'|z'b               500 Oops
```

The latter case is significantly harder to handle. To find such vulnerabilities we need to skip the quote-identification stage and guess the concatenation character to find evidence of a vulnerability, making the scanner less efficient. As we can't put random text in probes, we're constrained to a limited number of unique probes which makes reliably fingerprinting responses harder. At the time of writing the scanner doesn't handle such cases, although an early prototype has confirmed it's definitely possible.

This limitation doesn't apply to detecting injections into numeric inputs - given a base number, there is an infinite number of ways to express the same number using simple arithmetic. I've opted for x/1 and x/0, since dividing by zero has the added bonus of throwing an exception in some circumstances.

Recognising Significant Response Differences

The technical challenge at the heart of this technique is recognising when an application's response to two distinct probes is consistently different. A simple string comparison is utterly useless on real world applications, which are notoriously dynamic. Responses are full of dyamic one-time tokens, timestamps, cache-busters, and reflections of the supplied input.

When I approached this challenge three years ago, I used the intuition that responses are composed of static content with dynamic 'fuzzy points'. I therefore tried to use a set of responses to generate a regular expression by stitching together blocks of static content (identified using the longest-common-subsequence algorithm) with wildcards. For reasons of brevity, I'll only mention a small sample of the crippling issues with this approach. For a start, it's computationally intensive - the longest common subsequence implementation I used was $O(n^2)$; the time it took to process a response was proportional to the length of the response *squared*. The regular expressions were often so complex that scanning the wrong application caused a denial of service on the scanner itself. It also fails to account for applications giving drastically different responses which are difficult to regex together, and shuffling the order of response content. Even timestamps in responses raise difficulties, because parts of them by definition only change every 10, 60, or 100 seconds. Finally, it's extremely difficult to debug, as identifying why a particlar response doesn't match a 500-line regular expression can be tricky. Each of these problems may sound solvable, but my attempting to solve them is why this code wasn't released two years ago.

Instead, Backslash Powered Scanner uses the simpler approach of calculating a number of attributes for each response, and noting which ones are consistent across responses. Attributes include the status code, content type, HTML structure, line count, word count, input reflection count, and the frequency of various keywords.

The selection and delivery of probes is also crucial in minimising diffing problems. To differentiate between response differences due to non-determinism and differences caused by our probes, it's necessary to send each pair of probes multiple times. A scanner tha simple alternates between two payloads will fail and report false positives when confronted with an application that happens to alternate between two distinct responses, so be sure to mix up the probe order. Some particularly pernicious applications reflect deterministic transformations of user input, or even use user input to seed the choice of a testimonial quote. To remedy this, rather than probe-pairs we use pairs of sets of slightly different probes. Finally, caches can make 'random' content appear permanent, but this can easily be fixed using a cache buster.

Hunting Findings

Scanning Distributed Systems

Seeking to evaluate the scanner on real word systems and having a relatively limited supply of pentests, I decided to run it on every website within scope of a bug bounty program that doesn't disallow automated testing. This is a couple of thousand domains by my calculation. To display courtesy (and avoid being IP-banned), I needed to throttle the scanner to ensure it only send one request per three seconds to each application. Burp Suite only supports per-thread throttling, so I've coded and released an extension which will implement a per-host throttle. This extension also enables interleaving scan items on different hosts to ensure the overall scanner speed is still decent, and generating host-interleaved lists of unfetched pages for efficient throttled crawling. It also makes some other minor optimisations to improve scan speed without significantly reducing coverage, such as only scanning unpromising parameters like cookies once per host per response type.

Sample Results

MySQL Injection

I'll start with an easy example to show the context. This came from a site that was vulnerable to SQL injection via the User-Agent header:

```
Basic fuzz  (\z`z'z"\ vs \`z\'z\"\\)
    Content: 5357 vs 5263

String - apostrophe  (\zz'z vs z\\\'z)
    Content: 5357 vs 5263

Concatenation: '||  (z||'z(z'z vs z(z'||'z)
    Content: 5357 vs 5263

Basic function injection  ('||abf(1)||' vs '||abs(1)||')
    Content: 5281 vs 5263

MySQL injection  ('||power(unix_timestanp(),0)||' vs
'||power(unix_timestamp(),0)||')
    Content: 5281 vs 5263
```

Filtered Code Injection

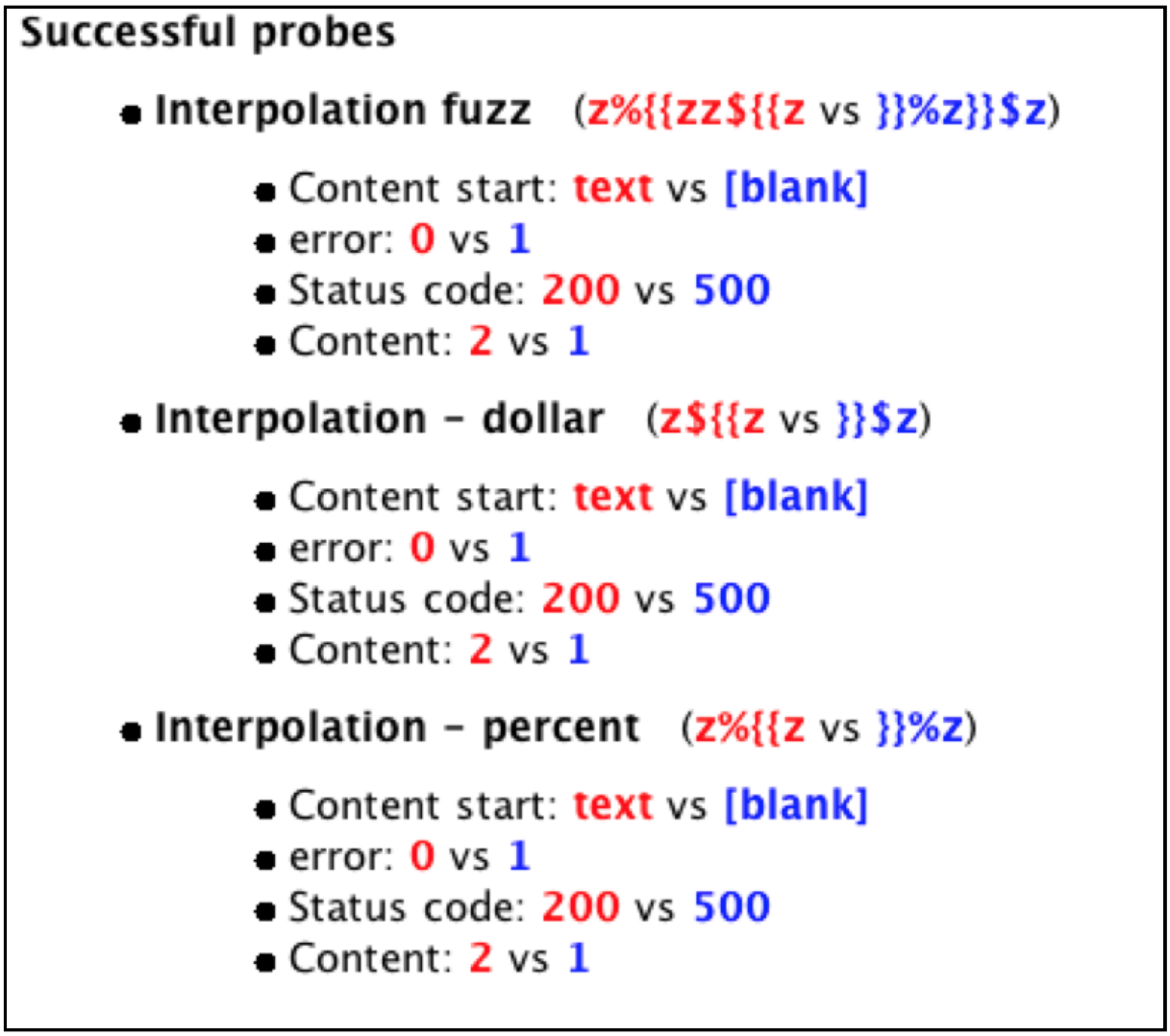
The folllowing finding comes from a pentest of a site that had already been tested numerous times, and clearly shows the power of this scanner:

```
String - doublequoted (\zz" vs \")
    error: 1 vs 0
    Content: 9 vs 1
    Tags: 3 vs 0
Concatenation: ". (z."z(z"z vs z(z"."z)
    error: 1 vs 0
    Content: 9 vs 1
    Tags: 3 vs 0
Interpolation - dollar (z${{z vs }}$z)
    error: 1 vs 0
    Content: 9 vs 1
    Tags: 3 vs 0
```

This was vulnerable to PHP code injection, but parethesis were being filtered out by the application - it's the second of the three blind spots of classic scanners mentioned earlier. I think the reason this vulnerability was missed by previous pentesters is that the injection was in the file path, which perhaps isn't somewhere a time-pressured tester would bother to check for code injection vulnerabilities. Why the application was calling eval() on the path remains a mystery. It's the kind of behaviour you expect from a internet of things device, not a household name website.

Old vulnerability

The following finding shows the current status of the input on sea.ebay.com that was previously vulnerable to PHP code injection. We can clearly see that the application responds differently to any input containing the { character.



Note that the responses demonstrate a behaviour opposite to what a naive fuzzer might expect - the string intended to break the application (${{z) causes a 200 OK response, whereas the harmless string causes a 500 Internal Server Error. Even though the search function is broken, the scanner has identified a clue of a vulnerability that used to be. Since the scanner is so efficient, it's perfectly plausible to try the PHP array-bypass attack on every input.

Regular Expression Injection

The scanner identified quite a few regex injection vulnerabilities, using both the input-transformation and diffing techniques. This is typically a low severity issue - it can be used to interfere with application logic and perhaps cause a denial of sevice (ReDoS) but little else. An exception is on servers running PHP<5.4.7, where regex injection can be escalated to arbitrary code execution by using a null byte to specify the 'e' flag[9]. This technique was recently used to exploit phpMyAdmin[10], and I've verified that the scanner finds it. Regex injection is typically reported with the following fingerprint:

```
Diffing scanner: Backslash  (\ vs \\) Transformation Scanner: \0 =>
Truncated
\1 => Truncated
\$ => $
$ => $
```

Backreferences like \0 offer a simple way to recognise regex injection. Applications may treat \99 differently from \100, and expand lower groups like \0 or \1 to other strings:

```
GET /folder?q=foo\0bar HTTP/1.1

HTTP/1.1 301 Moved Permanently
Location: https://redacted.com/folder/?
q=foohttp://redacted.com/folder/bar
```

Escaping Flaws

The scanner identified a cute but useless flaw in the way a popular web framework escapes values to be put into cookies:

```
foo"z: Set-Cookie: bci=1234; domain="foo\"z"; foo\: Set-Cookie:
bci=1234; domain="foo\"; foo"z\: 500 Internal Server Error
```

This framework proved so popular that I added a followup probe to automatically classify this issue and prevent anyone wasting time on it:

```
Basic fuzz  (\z`z'z"\ vs \`z\'z\"\\)
    exception: 1 vs 0
Doublequote plus slash  (z"z\ vs z\z)
    exception: 1 vs 0
```

Semantic False Positives

The function injection detection code raised a single false positive:

```
Function hijacking (sprintg vs sprintf)
<div: 13 vs 14
```

The root problem is evident from the URL: https://code.google.com/hosting/search?q=sprintg. The q input is being used to search a large codebase, where 'sprintf' is naturally a far more common term than 'sprintg'. Search functions are frequently ranked as interesting by the scanner, particularly those that support advanced syntax as they can appear deceptively similar to code injection vulnerabilities.

Web Application Firewall

Web Application Firewalls provide another source of 'interesting' behaviour. The scanner noticed that inline comments were being ignored on an otherwise value-sensitive input:

```
0/**z'*/ vs 0/*/*/z'*/
```

Manual investigation revealed that even HTML comments were being ignored... and also iframes.

```
0<!--foo--> vs 0<!--foo->
0<iframe> vs 0<zframe>
```

It looks like a Web Application Firewall (WAF) is rewriting input to remove comments and potentially harmful HTML. This is good to know - input rewriting effectively disables browsers' XSS filters. As ever, we can automate the HTML-comment followup to prevent this WAF from being a reoccuring distraction.

SOLR JSON Injection

The scanner flagged some interesting behaviour exhibited by a search function:

```
Basic fuzz (\z`z'z"\ vs \`z\'z\"\\)
    Content: 1578 vs 1575
Backslash (\ vs \\)
    Content: 1576 vs 1575
String - doublequoted (\zz" vs \")
    Content: 1578 vs 1575
```

Manual testing revealed that the application was decoding unicode-escaped input too - searching for \u006d\u0069\u0072\u0072\u006f\u0072 returned the same results as searching for 'mirror'. It appeared that user input was being embedded into a JSON string without escaping, enabling us to break out of the search string and alter the query structure.

Lessons Learned

These examples clearly show that the probe iteration process is crucial - it means that at a glance, we can distinguish a clearly critical issue from something that may take untold hours of investigation to classify. At present, search functions, WAFs and regex injections are a persistent source of promising looking behaviour that doesn't normally lead anywhere useful. Due to the flexibility of the probe-pair approach, almost every dud lead we encounter can be automatically classified in future with a followup probe.

We've also seen that the scanner can identify information that is useful even though it doesn't directly cause a vulnerability.

Many of these vulnerabilities were found on applications protected by WAFs - it appears that the simplicity of the payloads used makes them slip past WAFs unnoticed. However, I found that per-host rate limiting won't keep you off the radar of certain distributed firewall solutions that share IP-reputation scores. I managed to get the office IP banned from oracle.net without sending a single packet to it.

Further Research

The techniques and code used in the scanner can be adapted to detect far more than server-side injection vulnerabilities. We've already seen that followup probe pairs can be used to identify WAFs, and search functions.

Iterable Input Detection

Applications frequently suffer from access control bypasses where attackers can perform unauthorised operations simply by incrementing a number, for example on a URL like `/edit_profile?id=734` First, confirm that id=734, id=735, and id=736 return distinct responses. Fetching three distinct responses shows that the id input is being used, and that we're getting more than an 'invalid id' message. Next, we need to distinguish iterable inputs However, the application might just be performing a fixed transformation on the input or using it to seed an RNG. By requesting id=100734 and id=100735, and confirming they match, we can verify that we're retreiving data from a finite set.

Cold-start Bruteforce Attacks

Pentesters are often in a situation where they want to bruteforce a value, but they don't know what the success condition looks like. I made the earliest version of this scanner on a pentest where an ill-prepared client had failed to provide me with a single valid username, let alone a password. In order to stand a chance of guessing a valid password I had to bruteforce a username first, but the response to a valid username might be only subtly different, and I couldn't manually review thousands of login attempts. Using the simple response technique, this attack can be reliably automated. This approach can even bypass anti-bruteforce measures; when testing this tool I found that addons.mozilla.org gave a slightly distinct response to login attempts with a valid password, even when the account was locked due to excessive login attempts.

Bruteforcing file and folder names on servers that don't issue 404 responses raises a similar challenge. With sufficient logic, we could also use this technique to bruteforce hidden parameters to find mass-assignment vulnerabilities, and perhaps even bruteforce valid objects for deserialization exploits.

Conclusion

Classic scanners have several serious blind spots when it comes to identifying server-side injection vulnerabilities. By modelling the approach of an experienced manual tester, I have created a scanner that avoids these blind spots and is extremely efficient. It currently classifies inputs as either boring, interesting, or vulnerable to a specific issue. Issues classified as interesting require manual investigation by security experts, so at present this tool is primarily useful only to security experts. The scanner can be adapted to classify individual issues, so over time the proportion of issues classified as 'interesting' instead of 'vulnerable' should drop, making it more suitable for less technical users.

References

1. http://blog.portswigger.net/2014/07/burp-gets-new-javascript-analysis.html
2. https://www.blueclosure.com/product/bc-detect
3. http://blog.portswigger.net/2015/08/server-side-template-injection.html
4. http://artsploit.blogspot.co.uk/2016/08/pprce2.html
5. https://github.com/linkedin/dustjs
6. https://bugzilla.redhat.com/show_bug.cgi?id=1204307
7. http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html
8. http://secalert.net/2013/12/13/ebay-remote-code-execution/
9. https://bitquark.co.uk/blog/2013/07/23/the_unexpected_dangers_of_preg_replace
10. https://www.phpmyadmin.net/security/PMASA-2016-27/