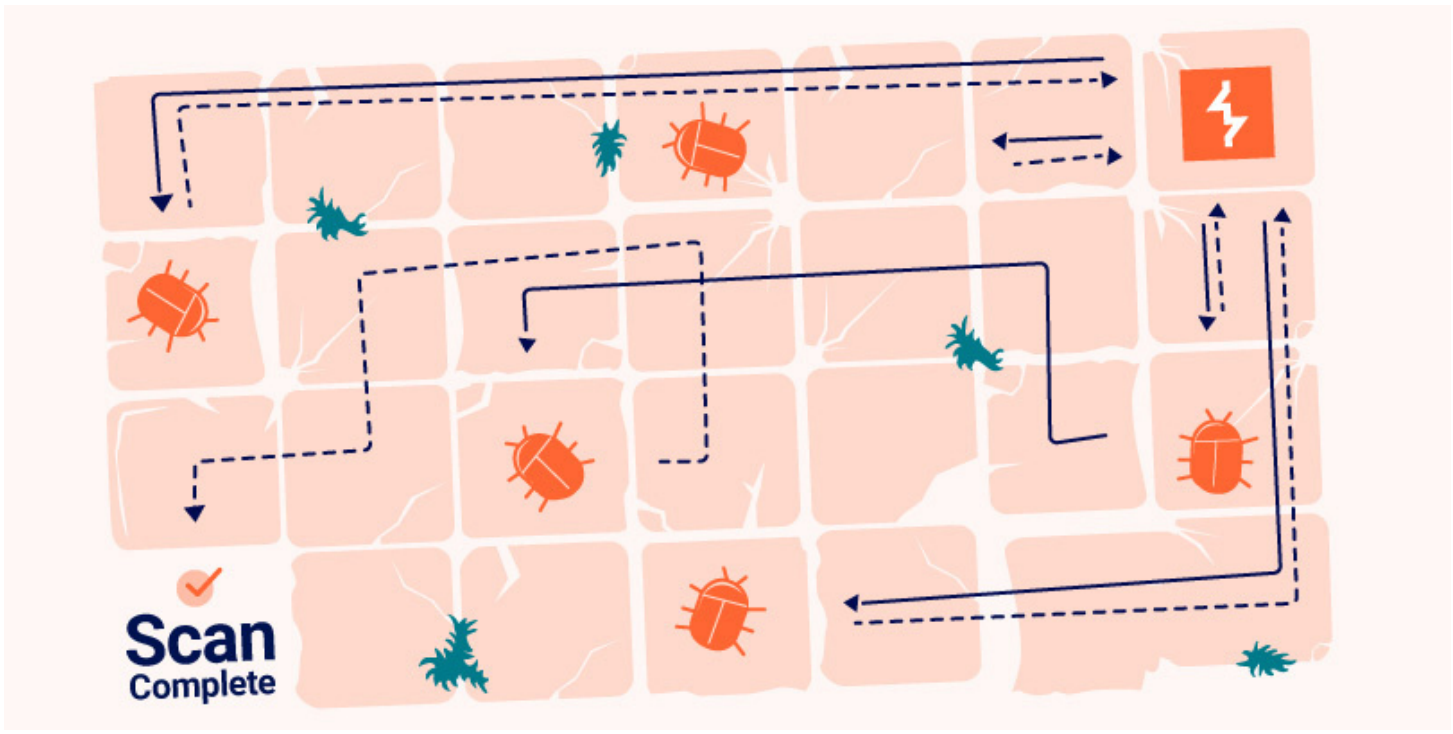
Web application cartography: mapping out Burp Suite’s crawler

Tom Shelton-Lefley | 05 March 2021 at 15:07 UTC

Burp Suite



At the core of Burp Suite is Burp Scanner - a powerful tool designed to reduce the number of manual steps users have to take to discover vulnerabilities in their targets.

Burp Scanner was first released back in 2008, and enabled users to quickly and reliably check for a wide range of standard issues - freeing up their time and expertise to be better spent on more interesting, novel bugs. After manually crawling their target and looking for points of interest, users could simply send their chosen requests to the scanner and get back an issue report covering common vulnerabilities (dozens at time of initial release, over 150 at time of writing).

However, as the web moved rapidly away from sites consisting primarily of static pages with simple content to highly dynamic, stateful applications, it became more and more difficult to get the most from Burp Scanner without first configuring manual session handling rules, macros, and other steps - eating back into the time originally saved.

Our goal when upgrading the scanner for Burp Suite 2 was the same as when it was first created: to remove the need for tedious, manual steps when searching for vulnerabilities. However, in addition to automating placing and sending each payload, this time around our focus was on seamlessly maintaining the correct session context for each of them, with no additional user steps.

This is where crawling comes in. It’s the crawler’s job to map out an application - not only finding all of the functionality which makes up the attack surface, but also how to reliably reach it. The crawler feeds items into the audit, and the audit burps out issues.

As one of the members of the scanner team at PortSwigger, who work on everything crawl and audit related, I wanted to share with you the challenges we faced when implementing the crawler for Burp Suite 2, some of the improvements we’ve made in the two years since, and finally what we have planned heading into the future.

The predecessor: Burp Spider

Burp Suite 1 came bundled with Burp Spider, a tool for content discovery which worked very well for very simple applications. The spider was effective, but simple - it followed these three steps:

1. Scrape the current page for links.
2. Add the links to the back of a queue.
3. Request the next link in the queue.

These requests and responses would be sent to the site map and could then be manually sent to the scanner. However, first the user would have to figure out whether the response relied on any session information. Then, if so, how to regenerate that session information, and finally record those steps as a macro. Failing to do this might result in the scanner logging itself out halfway through, and continuing oblivious.

When considering how to minimize the manual steps involved in scanning, the spider was part of the problem - it was time for an upgrade. We had the technology. We could make it better than it was. Better … stronger … faster. Okay perhaps not faster but I’ll be going over why that is.

Drawing a five dimensional map

Imagine you’re a cartographer tasked with mapping out a maze composed of rooms and doors. Now imagine the maze doesn’t make any sense: the same doorway doesn’t always lead to the same place, going backwards might take you forwards, the wallpaper changes every time you blink. In a broad sense, this is what the crawler is dealing with.

To tackle this problem, we broke it down into three main parts: knowing where you are, knowing where to go next, and knowing why things have changed.

Knowing where you are

When crawling a web application, being able to correctly identify where you are and recognizing where you’ve been are essential for knowing when you have discovered new content, and reliably finding your way back to it. Being able to identify a location not only allows you to say with some certainty that you are where you expect to be, but also to trim new areas which you realize you’ve already explored.

Two common features of modern web applications which complicate this are volatile content, and reusable templates:

- “Volatile content” refers to pages where subsequent visits to the same page return responses which differ substantially, for example due to adverts, or user generated social feeds.
- “Reusable templates” refers to multiple different pages returning responses built around the same set of core content, for example product pages on an e-commerce site, or posts on a blog.

Multiple visits to this “my account” page contain different “popular products” links, but the core of the page remains unchanged.

The crawler attempts to solve this problem by constantly refining its understanding of the core content in each location. It works on the assumption that, if it is taking steps through an application that it has already taken before, then the locations along that path should remain the same. If, when taking a step the crawler has already taken, the response differs slightly from those previously seen, then it can loosen constraints around the content which has changed. If they

differ too much, then it concedes that there has been a change and looks to see if that change is the result of session or application state changes.

We made two choices which allow us to use this assumption:

- The crawler explores the application using a breadth-first approach. This means that we often revisit locations as we increase depth to find new ones.
- The crawler always starts at the provided entrypoint for the application for every path, never "jumping" to a location out of context. Session state is also cleared. The means that each step is made with exactly the same context as when it was originally made, any differences are purely down to the application.

The crawler cannot begin on the "Our products" page to reach the pending content, it must first walk to that page from start before continuing on.

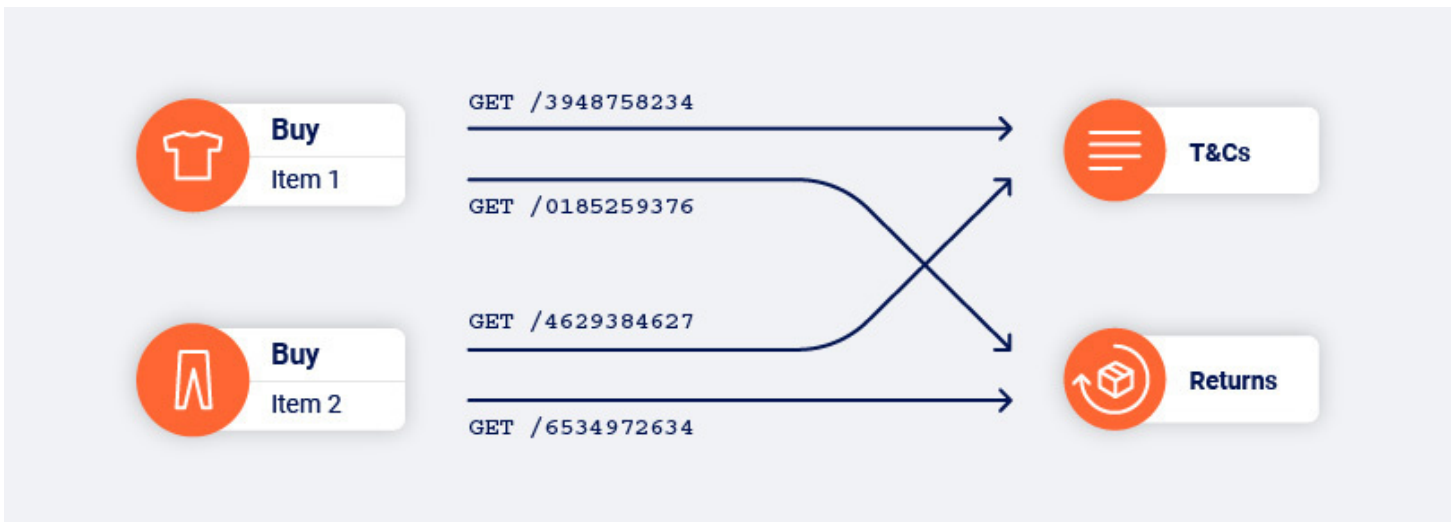
Knowing where to go next

Let's take a look back at the maze analogy at the start of this section - we now have a mechanism for knowing if the room we're in is one we've been in before. The next step is knowing which doorways we've previously walked through. This is important when we're retracing our steps.

As discussed, knowing which room we're in doesn't necessarily mean that the room is entirely the same. If features of the room have changed then that might include features which helped us enumerate the doors last time we were here. In order to be certain that the next step we take is the correct step for our planned path, we have to be sure we're using the same door.

Once again, volatility and templates are our adversaries here:

- If a page contains links with tokens or cache busters, then these links will appear different each time we visit the page.
- If many pages were built from the same template, then the links on them may differ to match the content injected into the template whilst still being functionally equivalent.



These two product pages were built from the same template. Their links look nothing alike, but they perform the same function.

The approach we took to this problem is much the same as the one we took for recognizing content features across a whole page. The crawler refines its understanding of what is core to each link and what is mutable with each visit. Different types of links use different strategies (anchors, forms, and so on) to ignore information which is proven to

change, and retain information which can be used to reliably identify a link over and over again.

We've covered how Burp Suite's crawler tackles identifying links (or doorways in our analogy) that appear different but map to the same function. However, we're also presented with the converse: links that look identical but map to different functions. You could think of this as being like synonyms and homonyms, respectively.

In this checkout flow, the structure of the "next" link does not change, but the function it performs does.

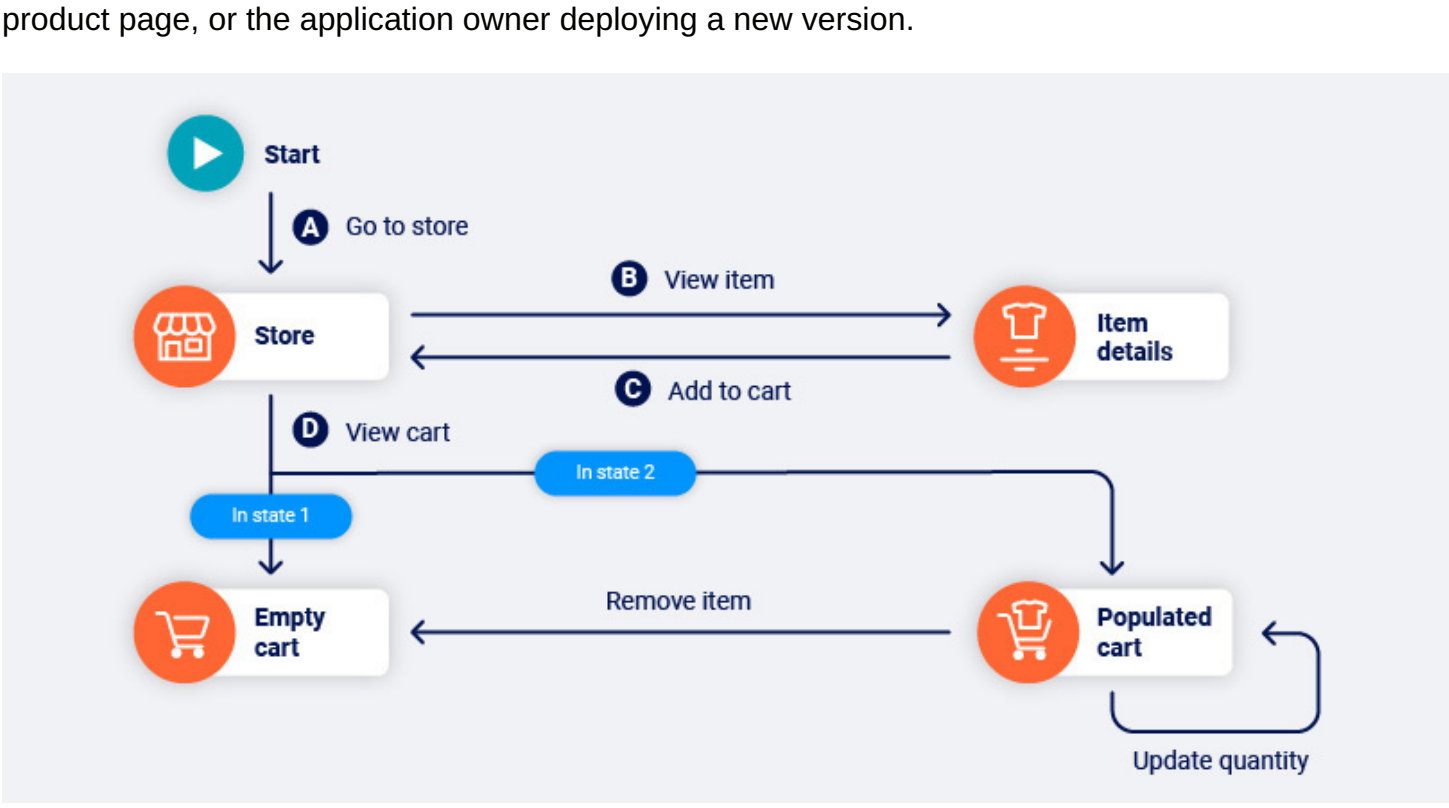
This latter case of homonym-like links is why it's important that the crawler does not use the incoming link structure alone as a factor when identifying where it is in the application. Really this point relates more to knowing where you are than it does to knowing where to go (even if we have a page filled with identical links, we can at least use their indices as a final feature to separate them). However, as it follows on nicely from discussing synonym-like links, I decided it's best to circle back.

Knowing why things have changed

At this point we've built a crawler which can move through our maze, reconciling small differences as it goes to keep track of where it is, and where it has to go next. But what about larger changes? What happens if we arrive in a room and it looks nothing like what we were expecting?

There are two circumstances in which this could happen: either we did something to change our session state, or something happened that changed the application state.

- An example of something that we might have done to change our session state would be adding a product to our cart on an e-commerce website. This action changes the cart page from an empty cart to a populated cart, with different available functions in each.
- An example of something that might have happened to change application state would be another user deleting a product page, or the application owner deploying a new version.



In this example e-commerce flow, visiting the cart before (AD) and after (ABCD) adding a product results in a different set of possible functions.

With this in mind, our first step when encountering a state change is to try to ascertain which of the two scenarios is the

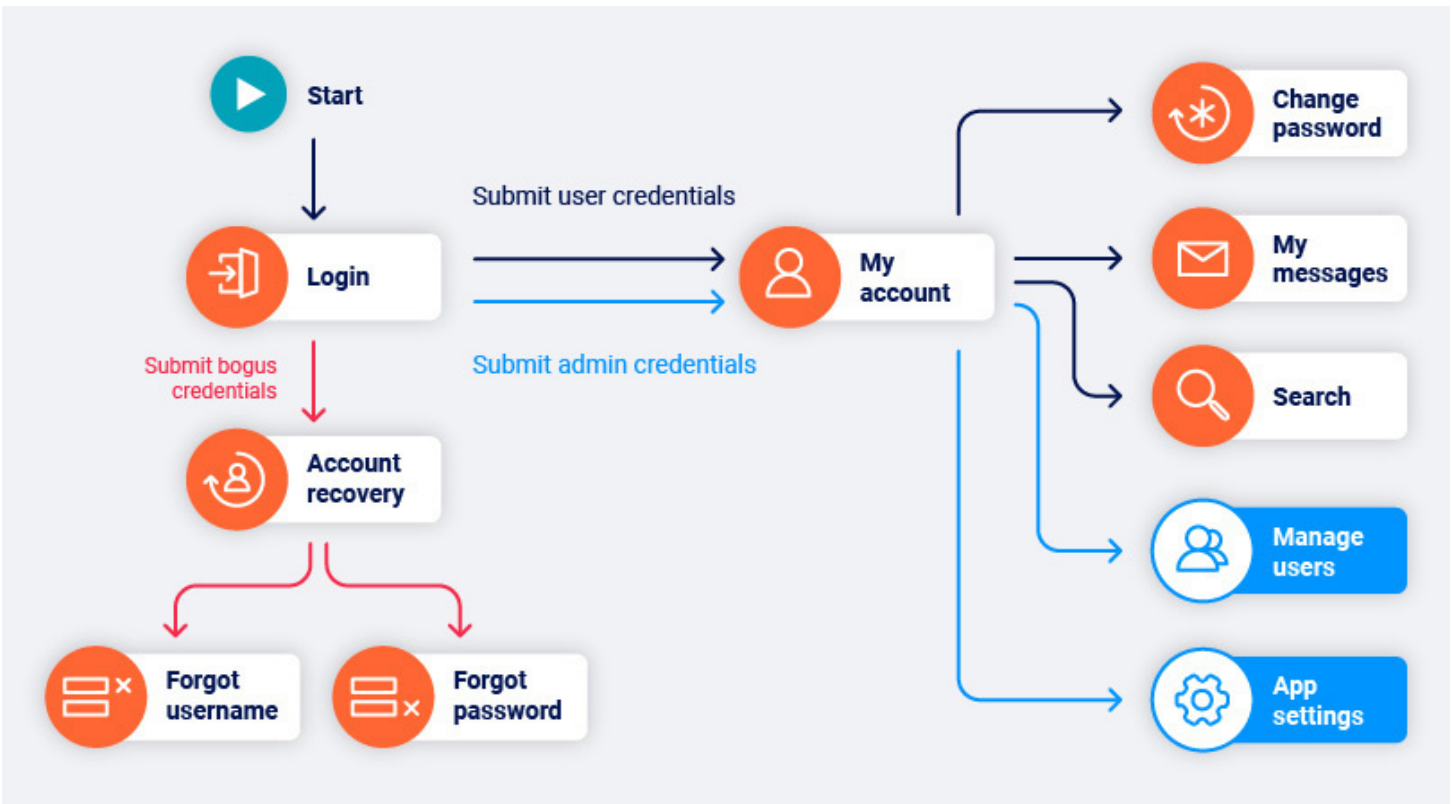
cause. If it's something we did, then we should be able to do it again. If it's a change in the application then we shouldn't be able to find the "old" version any more.

To achieve this we retrace our steps. The crawler re-walks every path it's recorded back to this location and compares the results:

- If all paths lead to the old version of the location then we mark the location as non-deterministic and, for want of a better term, "keep an eye on it" to see if it changes seemingly without reason again. If this does keep happening then the crawler just has to bear in mind that, to specifically reach either version of the location again, it may have to try several times.
- If all paths lead to the new version of the location then it is likely that the application has changed. We can safely replace the old version with the new version in our view of the application and continue.
- If some paths lead to the old version of the location, and others to the new, then we can assume that there is something we are doing in some paths and not others which is altering our session state. For all intents and purposes we treat the two versions as entirely separate locations.

I mentioned earlier that we clear all session state (cookies, for example) every time we start walking a path. This is crucial for the decision making flow above: if we weren't working with a blank slate for each path then we could not safely assume that any differences were triggered by the steps in that path.

The ability to detect and reproduce changes in session state is especially useful for applications with login functionality. We can identify which areas of the application are always accessible, which locations change when we're logged in, and which locations are only accessible to an authenticated user. This even extends to identifying application changes between different users with different permissions.



In this application, the locations beyond the login form change depending on whether the login was successful or not, and which user we successfully logged in as.

Putting it all together

Let's return one final time to our seemingly impossible maze. The issues we identified that we needed to overcome were:

- Going backwards doesn't necessarily lead us to the room we were previously in.
- The room behind a known door might not always be the same room.
- The same room or doorway might look different on subsequent visits.

By building a crawler that:

- Only ever walks forwards and always starts from the entrance.
- Rewalks paths to understand why rooms might have changed.
- Continually updates the important features of rooms and doorways with each visit.

Burp Suite's scanner can traverse this maze without ever getting lost. Most importantly, we can also plot an accurate map that can reliably lead us back to each location in the audit.

How this feeds into the audit

Back in the days of Burp Suite 1, if you had an application which made use of session state, or authentication, or dynamic requests (forms with CSRF tokens, for example) then chances are the scanner would fail to properly scan these locations. Worse than that, it would fail silently. The scanner had no notion of how an application should respond to a base request, so if the headers or tokens in the request became invalid at any point it would continue sending attacks regardless - with no indication that the desired part of the application was no longer being hit.

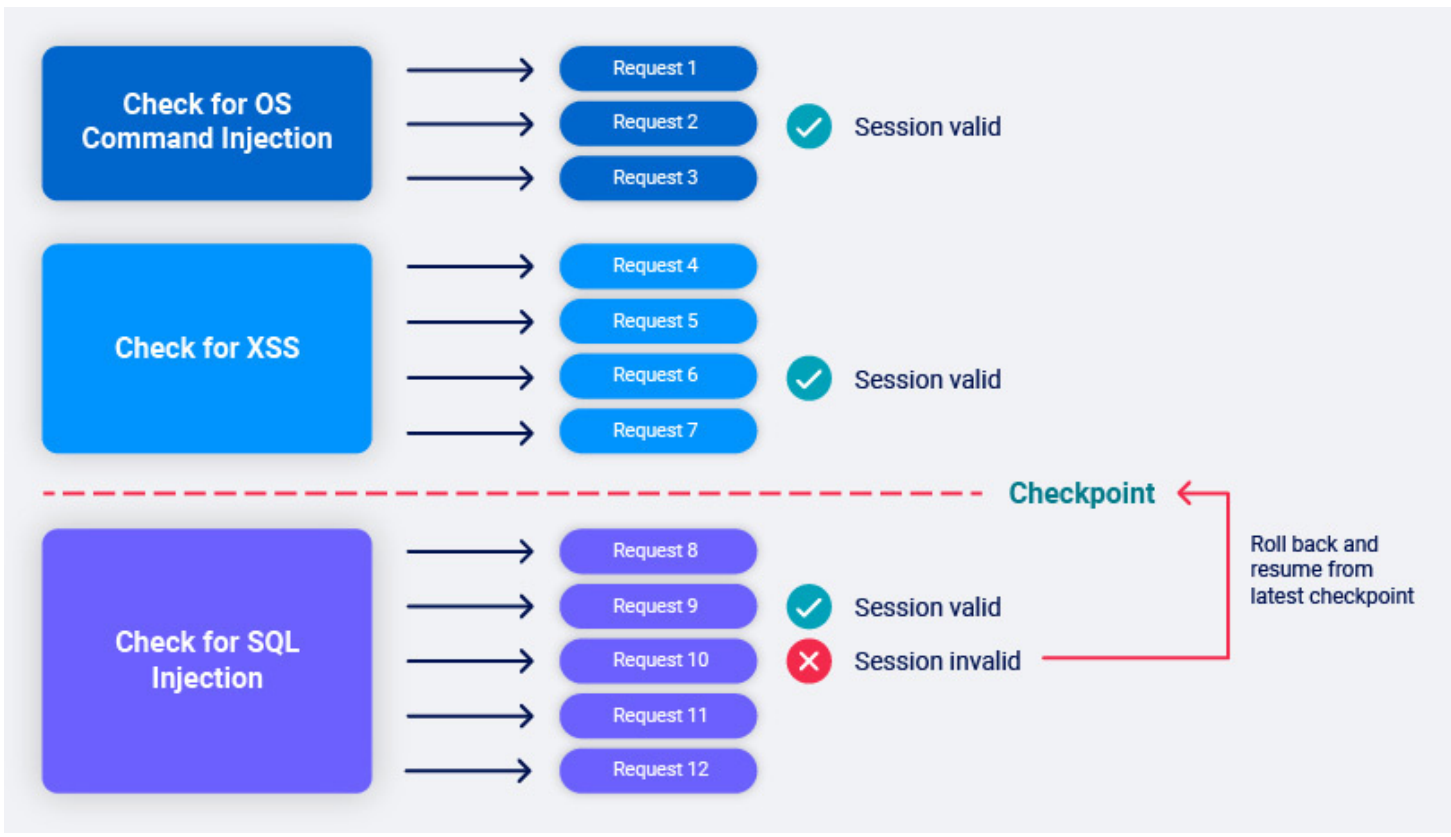
This could be overcome by setting up session handling rules and macros to wrap each attack with a series of steps to ensure that all of the required headers, tokens, and such were correct for every request. However this added both overhead to the user - having to configure these rules - and to the scanner - having to execute the rules even when it may not strictly be necessary, for example if a cookie needs refreshing but then remains valid for a certain amount of time.

You could think of the crawler as automating this configuration: determining the steps required before each attack request to ensure the request context is correct (the path in crawler terms). However, it goes even further.

Not only do we feed the path to each base request from the crawler to the audit, but also the information needed to determine whether the response is as expected. Throughout the crawl we've been building up a set of features for identifying each location, and we can use this in the audit to check that the response to a given base request matches the location we expect to find.

With these two pieces of information we can automatically build an optimized, session-maintaining flow around each payload in the audit:

1. First we walk the path to the base request.
2. Then we issue the base request and check that our assumptions about the response still hold.
3. Then we begin sending payloads for scan checks using just the base request. While we're in session there is no need to re-walk the full path.
4. At intervals we re-issue the base request and check the response to determine whether we are still in session.
5. If we find that we are still in session we set a checkpoint after the last scan check which was fully completed.
6. If we find that we are no longer in session we roll back to the last checkpoint, and return to step 1.
7. If we find ourselves constantly out of session then we fall back to walking the path for every request. This might happen if the request uses a one-time token, for example.



After request 10 we re-issue to base request and find that we are no longer in session. We roll back to the last fully completed scan check (XSS in this case) and resume from there.

Improvements since release

The new scanner, featuring crawl and audit phases, was released as part of Burp Suite 2 in mid-2018. Since then, we've continued to build upon and improve the crawler, and by extension the audit phase, focusing on performance and web application coverage.

Tuning

When we were first building the crawler for the release of Burp Suite 2, we focused heavily on making sure it could handle dynamic, stateful applications - that was the unique feature, so we wanted to get it right.

As users got their hands on the crawler, and we began to receive feedback, it became apparent that perhaps we'd tuned the default settings a little too far towards handling applications with heavy use of state and dynamic content (at the expense of performance against more static targets).

When you launch a new crawl, or crawl and audit, you're able to choose a "crawl strategy" either favoring speed or thoroughness, with the former being more suitable for static applications, and the latter for stateful applications. The settings underneath these presets control aspects of the crawl, for example how readily we'll match pages into the same location, and how many values of a form's drop-down we'll explore.

While we do expose these settings to be tweaked, it's still important that the defaults work well for the "average" application. Over time we've iterated on what we believe this to be, and fine-tuned our "normal" preset to give a better experience out of the box.

Browser-powered scanning

At launch, the crawler could already do plenty that the spider could not. However we were still using the same system for finding and parsing links on each page. So our coverage for more modern, dynamic applications hadn't yet really improved, just our ability to scan them without manual intervention.

Browser-powered scanning does exactly what it says on the tin: we use an embedded chromium browser to move around the target application. Using the browser, we can interact with links and render responses exactly as a real user would. This means that we correctly trigger any event handlers and build any JavaScript-added DOM elements - something that was never the case when we used to parse links only from the HTML returned in a page's response and build requests directly from them.

Long gone are the days when the web was mostly static, server-side rendered pages. Utilizing a browser in the scanner is the largest single step we've taken towards being able to properly navigate the modern web, bringing Burp Suite's crawler back to the cutting edge.

We'll be doing a deep dive into browser-powered scanning in the not too distant future.

Recorded logins

I've already suggested how powerful a state-managing crawler is for web applications with authentication functionality. The crawler can identify which user is needed to reach each location in the application, and automatically ensure that the user is logged in when attacking the location.

One thing that the crawler can't do (sadly) is correctly guess login details to use. Therefore, when we released the crawler, we included a mechanism allowing users to configure sets of credentials which would then be used when login forms were found during the crawl. The tricky part is identifying the "correct" login form.

Our heuristic for picking out a login form from all forms found is fairly simple and mostly effective one:

1. Filter out any forms which don't have a password type input.
2. Sort the remaining forms by number of inputs.
3. Choose the form with the fewest number of inputs (the more inputs a form has, the more likely it is to be a registration form).

We then fill the password field with the supplied password, and if there's a username field (identified as having the name attribute "username" or some variation) fill that with the supplied username (appending an email domain if the username field is of type email and the supplied username doesn't have one).

60% of the time, this worked every time. However, it's fairly easy to think of examples of where it doesn't:

- Logins that use a single sign-on mechanism.
- Logins that require more than two pieces of information.
- Multi-stage logins where the username and password are input on different pages.
- Login forms where the username field has the name "username" but in a language other than English.
- Logins that include dynamic information.

Rather than keep an ever-increasing backlog of specific tweaks that specific users needed (some of which may have broken previous submissions), we decided to give users the power to configure exactly what they need in the most generic way possible.

Recorded logins is another "does what it says on the tin" tool (naming things is hard). Users can record a sequence of steps via a PortSwigger browser plugin, which will then be played back at the start of every authenticated sequence. This means we can now support login mechanisms from any external provider, using forms with any number of credentials, spread across as many pages as is necessary, and in any language.

Unfortunately, because recorded logins repeatedly play back the same static sequence, the crawler does not yet have a solution for forms requiring dynamic information such as MFA, character-select inputs, or a CAPTCHA (we are a robot).

Recorded logins is one of the newest features we've released for Burp Scanner and we're still continually making improvements and tweaks based on user feedback. The web is big. You just won't believe how vastly, hugely, mind-bogglingly big it is. For every happy path there's an edge case, and our edge cases have edge cases.

We'll be doing a deep dive into recorded logins in the not too distant future.

Headed crawls

This starts with a shout out to any other software developers: if you're anything like me, when debugging a front-end test - the browser puppeting kind - the first thing you probably do is set the test to run in headed mode so that you can visually see what's going wrong. This same line of thought is where headed crawls came from. We know many of our users are experts, and wanted to give them a richer toolset for investigating crawler behavior.

When you run a browser-powered scan, Chromium is started up headlessly. By this I mean there is no browser window for the user to interact with, pages are rendered virtually. This both helps with performance, and usability. It's unlikely that Burp Suite scanner users want Chromium windows popping up and stealing focus all the time!

Allowing users to start crawls in headed mode - alongside the recorded login replayer - was implemented not to improve the scanner per se, but rather to help users understand its behavior and visually debug any issues they might encounter. Since releasing it late last year, we've already seen an improvement in the clarity of the dialogue we're able to have with users, and have squashed a few bugs that became obvious to users once they could see how the crawler operated on their specific applications.

API scanning

Up until this point I've talked about the crawler navigating web applications in terms of parsing and following links on pages. However, there are other ways of discovering content. Two examples would be a robots.txt page or a sitemap, and when we released the crawler support for parsing these was included. A third example, which we started supporting recently, are API definitions.

If your web application has a structured REST API, then it is common to also serve a definition of this API using a standard scheme. This allows consumers of the API to discover its functionality, and understand its behavior. OpenAPI is one such example of an API definition specification, and version 3 is what we chose as our first step towards utilizing API definitions in the crawler.

By parsing any API definitions we find (or are seeded with) during the crawl, we can build base requests to use in the audit which cover all of the defined endpoints. This allows us to scan an application's full REST API without the user having to manually configure all of the requests themselves.

Work on supporting further API definition schemas, and increasing support for OpenAPI's full feature set, is ongoing. We'll also be doing a deep dive into API scanning in the not too distant future.

What's next?

2021 has only just begun (even if it still feels like 2020 never came to an end) but we already have a whole host of features and improvements planned out. Some of them are public, and there are a few exciting ideas we're holding closer to our chests.

Usability

Over the last three years we've packed the Burp Suite crawler full of features and crammed in as many improvements

as we could manage. We're now taking a deep breath and diving into the usability of what we've created so far.

Over the coming months we'll be working internally, and with users, to make sure the crawler is well documented, well understood, and built on an intuitive core. Tasks we have begun discussing include:

- More blog posts and deep dives into specific features, challenges, and solutions in the crawler.
- Improving and simplifying the out-of-the-box workflow for starting and making use of crawls.
- Better exposure and documentation of the various settings which can be used to fine-tune the crawler for a specific web application.

Single page apps

Better support for single page applications is probably the feature we on the scanner team see requested most often. It's not quite hit dark mode numbers, but it's getting there.

What we mean by single page applications is web applications that are composed of views or navigational paths which do not necessarily require a document request and response to accompany moving from one view to another. These types of applications present an issue for the current iteration of the crawler, and by extension the scanner, for two main reasons:

- Navigation is often tied to non-traditional HTML elements using javascript event handlers, not just anchors and forms.
- Navigating from one "page" to another may result in no HTTP traffic, or only asynchronous resource requests.

Both of these features break core assumptions found in the scanner code dating back to Burp Suite 1. Removing them is a massive undertaking but something we're already well on our way to achieving. Stay tuned …

Final thoughts

A crawler engineer's work is never done. The crawler is the key to web application coverage in the scanner, and we're constantly devising and implementing improvements. Recently, most of our efforts have been focused on refactoring the core of the crawler to lay the groundwork for better single page application support - something we hope to get into your hands soon!

I hope that this post has given you some insight into the challenges faced by the crawler, and how we were able to build (and continue to build) a tool that solves them. This will be the first in a series of blog posts that dive into how specific aspects of the crawler, and Burp Scanner, function. Look forward to those over the coming months!

Burp Suite

Latest Posts

New: Burp Suite Enterprise Edition Pay as you scan pricing

25 April 2023

New: Burp Suite Enterprise Edition Unlimited pricing

30 March 2023

Burp Suite Enterprise Edition Power Tools:

Unleashing the power to the command line, Python, and more

21 March 2023