Challenges in the Automated Testing of Modern Web Applications

Nathan Hamiel
Gregory Fleischer
Seth Law
Justin Engler

Abstract

The attack surface of modern applications is a multi-faceted and fragmented environment. Current automated security tools have been found lacking in many areas when looking at the full scope of an application assessment. Most security professionals fill these holes using a variety of tools to cover as much of the attack surface as possible, including custom scripts, manual testing, and semi-automated testing. This approach puts extreme pressure on security professionals, as the testing effort may be difficult to duplicate between different assessments. This also forces the tester to recreate the test process for each assessment to ensure complete coverage when assessing an application's security posture. Addressing these deficiencies requires improved tools that evolve purely manual testing into semi-automated testing as well as providing support for the continued need of custom script creation.

Overview

It can be scary to think about how little of the modern attack surface many tools cover. There is no one best tool for the job, and on top of that, some tools don't do a great job at anything. Often in the hands of general users the capabilities and limitations are not even thought of during testing. Point, click, done. The attack surface of modern web environments as well as their protection mechanisms have become more complicated and yet many tools have not adapted.

There is certainly no shortage of vulnerabilities in modern web environments, but we should be looking beyond low hanging fruit at this point. In between fully automated scanners and manual testing lies a sweet spot for the identification of vulnerabilities. Some of the juiciest pieces of information are not found by vulnerability scanners but are found by humans creating custom tests. This is why semi-automated testing space is so important. All of this complicated blending of protection mechanisms, services, and RIA technologies means that moving in to the area of semi-automated testing can be fraught with failure. We detail how these failures can be avoided as well as provide a tool that solves some of these problems as well as provides analysis for your own tools and scripts.

Problems

Automated Testing versus Modern Applications

Automated testing tools continue to treat web applications as monolithic entities hosted in a single location with the application processing GET and POST requests from the user's browser based on form submissions and link navigation. However, modern web applications are no longer designed in this manner, and testing tools that primarily treat them as such miss many important issues. In some cases, these missed issues are not necessarily security vulnerabilities onto themselves, but the issues provide a greater understanding of the overall application design and underlying implementation. In assessments where time is limited,

identifying these issues early and an intelligent manner can allow for more focused and successful testing.

The following issues have been observed when testing modern applications using a variety of automated testing approaches.

Session Analysis

Current assessment tools contain databases of vulnerability signatures to classify application responses (think XSS or SQLi) as the application is tested in a runtime fashion. Both commercial (AppScan, WebInspect) and free (Burp Suite, Skipfish) scanners attempt to identify vulnerabilities within this single request and response framework. While this approach is useful for finding some vulnerabilities, it ignores certain classes of vulnerabilities that can only be identified when analyzing the full body of requests and responses.

One type of vulnerability ignored by most application scanners is Denial-Of-Service (DoS). Identification of CPU-intensive scripts and application pages is hard when analyzing pages based on a single request and response, but they are much easier to identify when analyzing a full sessions worth of requests and responses.

Application Architecture

The application architecture and development approach has evolved significantly in the last decade. Modern web applications are no longer simple CGI processing scripts. Instead, they have become complicated multi-layered applications with their own design patterns and implementation techniques. The reliance on advanced browser capabilities and web service interfaces is commonplace.

Automated testing tools have largely failed to keep up with these advances. Although most web scanners have moved beyond Nikto-style word list scanning, many automated testing tools still treat each web location as a server script with a set of form data elements and anchor tags. This outdated view of web applications leads to unnecessary and redundant testing as well as failure to fully explore and test application components.

One common approach employed in modern web application is the Model-View-Controller ("MVC") design pattern. By using a separation of concerns, applications can be constructed that offer multiple views of same underlying data and processing logic. Underlying frameworks such as Ruby on Rails and ASP.NET MVC allow for applications that provide enhanced content support, multiple device support, and "friendly" URL locations.

It is not uncommon to encounter modern applications that respond differently based on the value of "Accept" HTTP header. Based on type of content that the user-agent is willing to accept, the application may generate different return content or even use a completely separate authentication mechanism. For example, a web endpoint may generate an HTML page for a web browser, but XML for a web service invocation. This endpoint may be further overloaded to return JSON to service XMLHttpRequest calls generated by the HTML page itself. Or, the application may return the same underlying data as an RSS feed to anonymous user requests. Automated testing tools fail to consistently test variations in the "Accept" header.

With the rise of smart mobile devices, applications have returned to User-Agent sniffing to deliver content that matches the device characteristics. The iPhone in particular is often dealt with specially in order to provide content that renders appropriately. Applications also return different content for web search robots. Automated tools do not consistently perform tests to determine if different User-Agent values influence returned content. Frequently additional application locations and directories can be discovered through the use of alternate user-agent string values when crawling the application. There are still web applications that perform Internet Explorer only logic that may be missed if web spidering is performed using a custom User-Agent header string that does not match IE closely enough.

It is only a convention of the CGI specification and web server implementation that defined how URL path locations were traditionally treated. The PATH_INFO portion has long been used to pass data to CGI applications, but modern web applications have completely decoupled how data is represented in a URL request and how that is mapped into data to be processed by the application. Parameterized data values that were once passed as query string values are now encapsulated into SEO-friendly URL slugs. Although no automated testing tool can probably completely distinguish this pattern, it appears many have simply ignored this shift in how data values are submitted to the application. Automated tools often treat these data values as web-server directories and perform many unneeded tests.

Rich Internet Applications and AJAX

Standard testing tools assume that page interactions occur across the traditional paradigms of form submission (POST) and page retrieval (GET). A tool could crawl or spider a page by simple analysis of HTML forms and links. However, modern applications have moved on to perform advanced client-side calculations before carrying out a new page request. In many cases, data is refreshed live on the page via Flash or AJAX techniques, which could result in completely different page content without changing the actual URL used. Furthermore, page or data transitions can now be based on purely javascript-generated events, such as a JS-generated right click context menu on a web page.

Some tools have begun to match javascript patterns to attempt to detect transitions to new pages. However, this type of analysis is often inaccurate, as it relies on a “dumb” match to guess at the target page, where the actual browser client might combine information from multiple page sections and dynamic requests to generate a new request URL. RIAs, such as Flash and Silverlight, are still completely opaque to most tools. A smarter approach would be to completely abandon pattern matching on javascript strings, and instead actually execute the javascript directly. A tool with an integrated browser could follow the same javascript-based page and data transitions as a real user, and could potentially even generate click events in the browser to crawl the site. A highly-sophisticated tool might even be able to incorporate plugins into its browser, to allow Flash, Silverlight, and similar technologies (even PDFs) to be crawled via mouse click.

Furthermore, a smart, “Web 2.0”-aware tool may need to develop a completely different way of representing a site’s state transitions, as the traditional “sitemap” of folders and pages becomes increasingly irrelevant in a world of highly dynamic AJAX and RIA sites.

XSS Simulation

Most tools detect XSS attacks by injecting “known bad” javascript strings (usually variations on the alert() function) and unique identifiers into parameters, and then look for the injected pattern in the response. This could miss complicated filter-defeating injections that can be constructed via multiple inputs, and sometimes reports false positives by reporting an injection where the strings did not change, but are not executable. A smart testing tool with an embedded browser could simply “virtually render” responses and watch for alerts fired during the rendering and javascript execution.

Authorization Issues

Many application security problems stem from improperly allowing access to data beyond what user is authorized to see. This can happen when a page is supposed to be secured behind a login page, or even between two different authenticated users or roles. To find these issues, a tool could make two passes across the site, once with each authorization role. The tool could then provide a “diff”-style view of the pages which were viewable in one role or the other, allowing an intelligent auditor a chance to results that appear out of place. Some semi-automated tools have implemented this behavior, but support is still rudimentary.

Another authorization problem exists where a single record is viewable via a detailed records page. A malicious user can change the ID parameter (or similar) and see records for which he is not authorized. The application does not check to see if the current user is allowed to see a particular record, but relies on the user not tampering with the request parameters. This was one of the very first known flaws in web applications, but tools still cannot assist in the detection of this type of vulnerability. It is unlikely that any automated process will ever be able to conclusively find this vulnerability (as it requires understanding of the meaning of the data), but a tool could look for patterns in pages that appear to function in this way, highlight them for a human tester, and provide the data in a convenient format to help a tester decide if the page is operating as intended.

Web Services

Modern applications increasingly rely on (or provide) a web services layer (usually XML/SOAP or JSON/REST, but others also crop up). Modern semi-automated tools, however rarely handle web services any differently than another page request/response, and can often miss the parameter values present in a web services request. Furthermore, few manual tools are available at all to explore or tinker with web services, and the few that are available are aimed at developers, not security testers, and often do not contain functionality useful to security testing.

Application Locality

Many automated web scanners fail to properly account for DNS usage in modern applications. It is no longer the case where an application is hosted in a single location or behind a single IP address. Application hostnames can map to many IP addresses through the use of Round-Robin DNS and geographic load balancing. During the course of an assessment, an automated tool may either be testing across many hosts or stick with a single host based on IP address. There is often limited visibility into which IP address for a given application is being tested. This can make testing challenging if vulnerabilities are present in one instance of the application, but

not in the other. Even if a automated scanner cannot fully test this, tools rarely provide notification that this is the situation.

When geographic load balancing is used, people in separate locations could be accessing two complete different application instances. In some cases a tester is not going to be even aware of this situation. Unless special care is taken to resolve the web server host name from multiple world-wide network locations, this could represent a significant blind-spot in purely blackbox application assessments.

The increased use of content distribution networks (CDNs) and third-party metrics collection sites have complicated automating testing. In many cases, these sites are not within the scope of application testing, and most be specifically excluded from automated scanning. But it is not clear that vulnerabilities in the static content served by these sites are properly accounted for when the site has been excluded from scanning.

Application Composition

Automated scanners continue to treat web sites as monolithic applications, but modern web applications can be comprised of a variety of sub-components all hosted behind a single exposed web site instance. It is common to encounter web application gateways that route and re-route requests to a multitude of back-end application server instances. Common examples are the use of load balancers, reverse proxies such as mod_proxy/mod_rewrite, single sign-on portals, and application connectors such as AJP. These devices usually make routing decisions based on the leading path value, but more complicated schemes have been observed.

Many automated tools fail to account for this situation, and provide limited ability to properly tune testing patterns and coverage. A scanner may provide a configuration option to specify whether the application is a traditional web application or a web service, but modern applications often combine these into a single hosted instance differing only by request location on the site. Whether the application and web service are implemented within the same physical application or routed to different back-end application server instances, an approach that attempts to split these into two separate tests usually misses important contextual information that may reveal significant vulnerabilities.

Something as simple case-sensitivity can cause issues with many scanners. Automated scanners construct tests that account for case sensitivity in order to reduce scan time and provide appropriate coverage. But instead of detecting case on a per-virtual path basis, the scanner will determine case on a per host name basis. In a composite application that incorporates multiple web server types, this can lead to unnecessary testing or missed tests. For example, consider a composite application that incorporates a ASP.NET web forms application and a RESTful web-service hosted on Tomcat. The request to these applications are transparently routed based on path location through a single reverse proxy web to separate internally hosted application server instances. To the outside world, this appears to be a ASP.NET application hosted on Windows. If the scanner makes an assumption about the overall case handling of the application based on Windows IIS handling, it may miss directories and file tests if it treats everything as case insensitive. A test against “changePassword” will be ineffective if “changepassword” is used instead.

Or, it may be the case where there are multiple applications hosted in a clustered server frame behind a single external IP. Internal host destination is controlled by a load-balancer that uses long lasting “sticky” sessions, but where the internal host is exposed as a cookie value. An automated scan produces no issues, but repeated manual testing finds that vulnerable files on a single cluster member. Once again, an automated tool may not be able to fully test for this situation, but only by manual review of other unrelated results could potentially reveal the issue.

Application Deployment

Many automated scanning tools continue to treat application testing as a point in time event. Once an application location is tested, it is never revisited during the course of the scan. However, many modern web applications are constantly being deployed through the use of continuous integration techniques. It is common to see applications in highly dynamic environments to be changed during the course of a long running scan. High availability sites use zero downtime upgrades to constantly roll out changes across large numbers of web hosts. This can have a direct impact on scan results.

It is surprisingly rare to find tools that fully take this into account. Often a tool will detect a new directory or page being added, but the scanning does not extend to changes to occur during the course of the scan run. Obviously, it would be infeasible to re-test every location to determine if a change has occurred, but in most cases, the ratio of issues found compared to the location tested is small. A reasonable approach would be to re-test any issues and report on discrepancies. Additionally, a small sample set of locations could be occasionally revisited to detect underlying changes to the application through redeployment.

Ideally, an automated tool would be able to use a generalized approach to continuous re-testing that could help detect changes in composition and locality that may manifest themselves through changes in DNS and cookie values associated with load balanced environments.

Content Discovery and Context

Often, automated testing tools fail to make intelligent use of discovered site information. “Google hacking” has been a staple of all manner of testing for years, but few scanners offer the ability to automatically take advantage of the search results information to populate additional scan locations. The use of Google, Bing and Yahoo! search results continue to offer value even as the search engines move to more restrictive models of interactions in order to affect maximum monetization. In cases where the Terms of Service of the search engine represent a legal gray-area, testing tools make few accommodations to import even manually retrieved results.

The last several years have seen an increase in the number of utilities that exist to fingerprint web applications and application versions, but scanners do not make use of this information in any meaningful way. If a fingerprint routine can determine a particular version of a Content Management System (CMS), why not perform SQL injections testing based on the tables in that version instead of generic ones?

Valuable information can be “leaked” by the application in page response, HTTP headers and HTML and JavaScript comments, but these data points are not used to make more intelligent

testing decisions. Sometimes, automated scanners make extremely poor use of contextual information. One more than one occasion it has been observed that a scan will detect a “icons” or “small-icons” directory on Apache with full directory listing enabled. The scanner then proceed to fuzz each newly “discovered” image file location with a variety of attack payloads. Besides the futility of that effort, the scanner failed to report on the truly valuable information which was the names, times and sizes of the images files. Only by manually reviewing this information was it possible to determine additional vulnerable software running on the server.

Another common weakness in automated scanners is the failure to classify aggregated content. Modern web applications tend to use a variety of third-party script and application content in addition to any custom code. Commonly observed libraries include jQuery and YUI. But many scanners treat this content identically and will include these directories and files in subsequent tests.

Some scanners detect email addresses that are embedded in site content. In many cases, this email information could be valuable for a variety of penetration tests, but too often the results are populated with email addresses from non-related content. Instead of useful information, the list of email addresses includes the authors of popular open-source JavaScript libraries. This information could be used to help identify installed versions of software, but that data is better presented at a higher level instead of polluting the legitimate results.

Payload Selection and Coverage

Web vulnerability scanners tend to favor completeness over coverage as some sort of best case approach. Unfortunately, in the real world, scans are sometimes unable to be run to completion with even reasonable test configuration and tuned options turned on. Frequently there is not enough available time to complete scans due to limited engagements or restrictive scanning windows.

When testing sites, especially ones with a large number of issues, the vulnerabilities tend to cluster together or are repeatedly observed across the site. If the goal of the assessment is enumeration of all vulnerabilities, it may make sense to extensively test each element on a form for variants of the same issue. But if the goal is breadth of coverage, many scanners fail to offer appropriate tuning for this.

In highly dynamic web applications, simply testing a registration form could involve tens of thousands of requests with varying payloads. If one input on a form is vulnerable to XSS or SQL Injection, it is more probable that other elements on the same form are vulnerable. In a time limited engagement, it may be more beneficial to move onto other forms instead of continually retesting the same form for different variations.

For example, in a page that is dynamically re-populated based on previous form choices (such as a state and country code drop-down), some scanners become stuck repeatedly re-testing and re-discovering the same issues as the list of available combinations are tested. Finding the same XSS in the email address for multiple combinations of state and country codes is hardly beneficial.

In the case of aggregated third-party content such as third-party script libraries, automated testing often performs poorly. Depending on how compact the content is, an unintelligent scan approach could take an excessive amount of time. Some YUI library distributions are especially prone to this as the list of directories and files is repeatedly tested. This behavior can be especially bad if directory indexing is enabled on some directories. Although there is some probability that these third-party libraries could introduce a vulnerability, repeatedly testing the script locations is unlikely to produce useful results.

Storage of Requests and Responses

Although commercial tools do offer the ability to export vulnerability information, the underlying request and response data is rarely accessible in a reasonable format. In some instances, this information is not stored at all. This is especially surprising, because disk space has become cheap, but network time and application processing continue to be expensive. Too often valuable request and response information is being thrown away instead of being stored for later analysis if needed.

For example, an application being tested returns a unique error code token in response instead of error message. The automated scanner that is being run generates these errors, but does not recognize the responses as errors due to the lack of error messages matching a predefined list of strings. The scanner discards these requests and responses and with them the error codes embedded in the returned responses. Later, manual testing finds exposed AJAX functionality that translates these error codes to messages. But the list of error codes has been lost and must be regenerated through the use of custom tool or utility to either re-request the pages or to re-run the scanner and capture raw network traffic which itself must be analyzed. These options are both equally undesirable.

Every automated tool is going to miss issues, but in general, it is easier to mine existing saved request and response data than it is re-request the information. In some cases, it may not even be possible to regenerate the requests if testing was performed in a staged environment.

Client-side Storage Locations for Sensitive Data

Automated testing tools do not account for data exposed in alternate client-side storage locations. The use of Flash Local Shared Objects (“LSO”) has long been a mainstay technique in persisting when browser cookies are cleared. With the gradual implementation of HTML5 storage features, additional storage on the client may be populated with sensitive data. Unfortunately, even manual testing tools offer limited visibility into these data values.

Modern web applications continue to populate sensitive data values in hidden form fields and cookies, but automated testing often fail to fully detect these values based on the their usage and data type characteristics. The ability of testing tools to detect the use of modern storage is almost nonexistent. Partially, this can be attributed to the fragmented nature of the standards. Various locations and technologies have been produced, implemented, and in some cases, discarded: WebStorage (“localStorage”, “sessionStorage”), WebDatabase, and IndexDB.

Combined with other browser specific technologies, such as Internet Explorer's "userData", it can be challenging to fully identify the scope of client side storage location.

In some cases, developers implement client-side storage using abstractions layers such as jStorage without fully understanding the implications of how the data will be persisted locally. Client-side storage values are dynamically populated with user identifiable information or other sensitive tokens. Client-side storage in Flash LSOs and to a lesser extent Silverlight's Isolated Storage as well as HTML5 storage locations are not visible to users and may be overlooked even during an in-depth manual review.

Even if automated testing tools cannot completely test values stored in these locations, simply identifying an application's usage of these techniques would be beneficial for later follow up.

Security Features without Good Coverage

In testing tools, there is a natural lag time between when new security features are defined and when they are rolled out in automated testing routines. This is especially true of commercial tools. Even then, some of the of the more technically oriented security controls continue to have spotty coverage in automated tests. Specifically, the "X-Frame-Options", "Origin" header, "X-Content-Type-Options: nosniff", and Content Security Policy are underrepresented.

When there is unneeded delay between the definition of new security standards and their integration into testing tools, it makes those tools seem outdated and unresponsive to innovations. It is an unfortunate fact that, in many enterprise environments, some of the these application level security issues are only dealt with as those issues begin showing up regularly in scan reports. For those organizations without dedicated application security staff, an automated scan report can be better than nothing in helping to expose new security techniques.

Several future changes are on the horizon that will challenge existing tools. The introduction of IPv6 represents a hurdle to tools that are under-prepared to deal with these changes. Tools should be reviewed to determine if differences in syntax and storage of IP address values will impact testing visibility. For example, tools that depend heavily on regular expressions to support link extraction may be impacted if the IPv6 literal address format is not properly accounted for.

Additionally, the last several years have seen increased attacks on SSL and a variety of proposals to deal with the identified deficiencies. The manner in which tools choose to deal with standards such as DNSSEC and proposed approaches such as DANE have a direct impact on how and when applications begin integrating those technologies.

Solutions

So what happens when you run in to a condition where none of the current testing tools support the testing situation? The obvious and most painful thing is to test for conditions manually. Testing manually is not only time consuming but fraught with error. Just the sheer amount of data that would have to be gone through in order to get a complete picture during a test is mind boggling.

Going down the road of writing your own scripts can allow for a certain amount of success in testing difficult applications, but typically there is no standard framework when this route is taken. These items are mostly used for one-offs and reuse isn't always a consideration. Often there is still the problem of analysis of the captured data. It is rare to write a script for testing that has robust analysis built in.

It may be possible to modify an existing tool to perform the test cases you need. For example a framework like Selenium or Windmill could potentially be modified to drive the browser through test cases. Often in these situations you are using tools that aren't specifically made for security testing, which can create a different set of problems.

Contributing to an existing open source tool can be a rewarding experience, however there are problems with this approach as well. Often existing open source tools are created for a specific purpose and then an expanded to include additional features. This means there may be no defined framework for making additions to the tool. You may also have to work with the developers of the tool as well which could offer up some challenges on that front.

Finally you can write your own tool or framework. This can present a ton of work, but also a large reward. You can define work flow as well as enhancements and capabilities. You also get intimate familiarity with with the programming languages and technogy used for the tool. This is the route we decided to take.

Our Solution

We decided to write our own tool called Response Analysis and Further Testing (RAFT). We decided to write a tool from scratch because we wanted to focus a bit on the work flow aspect of the testing process. Often when trying to integrate with another project the work flow and framework are already defined. It can often be difficult to get these items changed after a project has been introduced to the community.

RAFT is a tool that focuses on semi-automated testing and analysis of captured data. Our framework also includes a browser object that can be used for various tasks such as rendering pages or even interfacing with the DOM pragmatically. This allows us flexibility in difficult testing cases such as a randomized DOM environment.

The tool can also be used to run analysis on data captured from other tools or even from your own scripts. It is easy using RAFT to create your own custom analyzers to modify the tool to fit your needs. RAFT also has a fuzzer that can handle randomized data such as CSRF tokens.

RAFT can be downloaded here: http://code.google.com/p/raft

Appendix A: RAFT Capture Format

The RAFT Capture Format is an XML data format supported by the RAFT tool for import and export of request and response data. Although some tools prefer a binary format for storing data, an text-based XML approach is preferable for interoperability and its self-documenting nature. For binary content that cannot legally be represented as XML characters, Base64 encoding can be used to store the data in a reasonable compact and cross-platform manner. Although by its nature, XML data storage tends to be “chatty” and can balloon in size, the storage cost can be minimized by choosing an appropriate streaming compression mechanism such as lzma, bzip2 or gzip.

The DTD for the RAFT Capture Format version 1.0 follows:

```
<!-- RAFT Capture XML DTD version 1.0 -->
<!ELEMENT raft (capture*)>
<!ATTLIST raft version CDATA #IMPLIED>
<!ELEMENT capture (request, response?, analysis?)>
<!ELEMENT request (method?, url?, host?, hostip?, datetime?, headers,
body?)>
<!ELEMENT response (status?, content_type?, content_length?,
elapsed?, headers, body?)>
<!ELEMENT analysis (notes?, confirmed?)>
<!ELEMENT method (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT hostip (#PCDATA)>
<!ELEMENT datetime (#PCDATA)>
<!ELEMENT headers (#PCDATA)>
<!ATTLIST headers encoding (none|base64) "none">
<!ELEMENT body (#PCDATA)>
<!ATTLIST body encoding (none|base64) "none">
<!ELEMENT status (#PCDATA)>
<!ELEMENT content_type (#PCDATA)>
<!ELEMENT content_length (#PCDATA)>
<!ELEMENT elapsed (#PCDATA)>
<!ELEMENT notes (#PCDATA)>
<!ELEMENT confirmed (#PCDATA)>
```

These data elements have the following usage.

Element	Description	Example
raft	XML root element	`<raft version="1.0">`
capture	XML capture element	
request	XML request element	
response	XML response element	

analysis	XML analysis element	
method	Request method	<method>GET</method>
url	URL of request	<url>https://www.google.com/</url>
host	Host name	<host>www.google.com</host>
hostip	IP address of host	<hostip>74.125.225.69</hostip>
datetime	Date and time of request	<datetime>Sun Jul 10 21:13:26 2011 GMT</datetime>
headers	Raw headers value for request or response. The text content should have XML encoding applied unless it contains invalid XML characters. If text content contains invalid XML characters, it should be encoding as Base64, and the 'encoding' attribute should be set.	<headers>GET / HTTP/1.1 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/ 533.21.1 (KHTML, like Gecko) Version/ 5.0.5 Safari/533.21.1 Accept: application/xml,application/ xhtml+xml,text/html;q=0.9,text/ plain;q=0.8,image/png,*/*;q=0.5 </headers>
body	Raw body value for request or response. The text content should have XML encoding applied unless it contains invalid XML characters. If text content contains invalid XML characters, it should be encoding as Base64, and the 'encoding' attribute should be set.	<body encoding="base64"> PCFkb2N0eXBlIGh0bWw+PGh0bWw+PGhlYWQ +PG1ldGEgaHR0cC1lcXVpdj0iY29udGVudC10eX Bl IiBjb250ZW50PSJ0ZXh0L2h0bWw7IGNoYXJzZXQ 9VVRGLTgiPjxtZXRhIG5hbWU9ImRlc2NyaXB0 . . . dHRhY2hFdmVudCgib25sb2FkIixsKTtnb29nbGU udGltZXJzLmxvYWQudC5wcnQ9KGY9KG5ldyBE YXRlKS5nZXRUaW1lKCkpOwp9KSgpOwo8L3Njcml wdD4= </body>
status	Response status code	<status>302</status>
content_type	Response content type	<content_type>text/html; charset=UTF-8</content_type>
content_length	Length of response body	<content_length>226</content_length>
elapsed	Elapsed time of request in milliseconds	<elapsed>2073</elapsed>
notes	Any textual notes associated with the capture element.	<notes>The request was invalid.</notes>
confirmed	Boolean indicating if the capture element was a vulnerability.	<confirmed>True</confirmed>

An (abbreviated) example shows how capture data could appear.

<raft version="1.0">

<capture>
<request>
<method>GET</method>
<url>https://www.google.com/</url>
<host>www.google.com</host>
<datetime>Sun Jul 10 21:13:26 2011 GMT</datetime>
<headers>GET / HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/
533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5

</headers>
</request>
<response>
<status>302</status>
<elapsed>2073</elapsed>
<content_type>text/html; charset=UTF-8</content_type>
<content_length>226</content_length>
<headers>HTTP/1.1 302 Found
Location: https://encrypted.google.com/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie:
PREF=ID=f01152de550f89b6:FF=0:TM=1310332408:LM=1310332408:S=f87Qj36iuMfjhII2;
expires=Tue, 09-Jul-2013 21:13:28 GMT; path=/; domain=.google.com
Date: Sun, 10 Jul 2011 21:13:28 GMT
Server: gws
Content-Length: 226
X-XSS-Protection: 1; mode=block

</headers>
<body><HTML><HEAD><meta http-equiv="content-type"
content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="https://encrypted.google.com/">here</A>.
</BODY></HTML>
</body>
</response>
</capture>
<capture>
<request>
<method>GET</method>
<url>https://encrypted.google.com/</url>
<host>encrypted.google.com</host>
<datetime>Sun Jul 10 21:13:28 2011 GMT</datetime>
<headers>GET / HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/
533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5

</headers>
</request>

<response>
<status>200</status>
<elapsed>1788</elapsed>
<content_type>text/html; charset=UTF-8</content_type>
<content_length>33890</content_length>
<headers>HTTP/1.1 200 OK
Date: Sun, 10 Jul 2011 21:13:30 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Set-Cookie:
NID=48=eQkZZpXtrs9dHIQ7KmhaTAegfKjVMXyrx5a41aIYXUJiktP9Awtj17BnjGK41riY839DGp
dODA8vC7QKVMP4bvQYEQ_ztwZx0XT4oMcnhowjptBkpMtvFl3t6kNiVnGy; expires=Mon, 09-
Jan-2012 21:13:30 GMT; path=/; domain=.google.com; HttpOnly
Content-Encoding: gzip
Server: gws
X-XSS-Protection: 1; mode=block

</headers>
<body encoding="base64">
PCFkb2N0eXBlIGh0bWw+PGh0bWw+PGhlYWQ+PG1ldGEgaHR0cC1lcXVpdj0iY29udGVudC10eXBl
IiBjb250ZW50PSJ0ZXh0L2h0bWw7IGNoYXJzZXQ9VVRGLTgiPjxtZXRhIG5hbWU9ImRlc2NyaXB0
. . .
dHRhY2hFdmVudCgib25sb2FkIixsKTtnb29nbGUudGltZXJzLmxvYWQudC5wcnQ9KGY9KG5ldyBE
YXRlKS5nZXRUaW1lKCkpOwp9KSgpOwo8L3NjcmlwdD4=
</body>
</response>
</capture>

<capture>
<request>
<method>GET</method>
<url>https://encrypted.google.com/blank.html</url>
<host>encrypted.google.com</host>
<datetime>Sun Jul 10 21:13:30 2011 GMT</datetime>
<headers>GET /blank.html HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/
533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
Referer: https://encrypted.google.com/

</headers>
</request>
<response>
<status>200</status>
<elapsed>266</elapsed>
<content_type>text/html</content_type>
<content_length>0</content_length>
<headers>HTTP/1.1 200 OK
Content-Type: text/html
Last-Modified: Fri, 13 Aug 2010 03:36:08 GMT
Date: Wed, 06 Jul 2011 18:14:04 GMT
Expires: Thu, 05 Jul 2012 18:14:04 GMT
X-Content-Type-Options: nosniff
Server: sffe
Content-Length: 0
X-XSS-Protection: 1; mode=block

```
Cache-Control: public, max-age=31536000
Age: 356366

</headers>
</response>
</capture>

<capture>
<request>
<method>GET</method>
<url>https://clients1.google.com/complete/init?tch=4&amp;amp;ech=1&amp;amp;psi=-
hUaTo6PD4LIqAGf35SgCg.1310332415279.1&amp;amp;wrapid=tljp131033241527920</url>
<host>clients1.google.com</host>
<datetime>Sun Jul 10 21:13:35 2011 GMT</datetime>
<headers>GET /complete/init?tch=4&amp;amp;ech=1&amp;amp;psi=-hUaTo6PD4LIqAGf35SgCg.
1310332415279.1&amp;amp;wrapid=tljp131033241527920 HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/
533.21.1 (KHTML, like Gecko) Version/5.0.5 Safari/533.21.1
Accept: */*
Referer: https://encrypted.google.com/

</headers>
</request>
<response>
<status>200</status>
<elapsed>1640</elapsed>
<content_type>text/javascript; charset=UTF-8</content_type>
<content_length>250</content_length>
<headers>HTTP/1.1 200 OK
Content-Type: text/javascript; charset=UTF-8
Date: Sun, 10 Jul 2011 21:13:37 GMT
Pragma: no-cache
Expires: -1
Cache-Control: no-cache, must-revalidate
Content-Disposition: attachment
Content-Encoding: gzip
Server: gws
X-XSS-Protection: 1; mode=block

</headers>
<body>window.google.td &amp;amp;&amp;amp; window.google.td('tljp131033241527920', 4,
{e:"ARYaTo-BAsaQsAKPsbTCBw",c:0,u:"https://clients1.google.com/complete/init?
tch\x3d4\x26ech\x3d1\x26psi\x3d-hUaTo6PD4LIqAGf35SgCg.
1310332415279.1\x26wrapid\x3dtljp131033241527920",d:""});</body>
</response>
</capture>

</raft>
```