Active Exploit Detection

Marc Eisenbarth
HP TippingPoint

January 7, 2011

1 Introduction

Security professionals have a massive number of acronyms at their disposal: IPS, VA, VM, SIEM, NBAD, and more. This talk is about a tool that resists classification by these acronyms. The goal of Active Exploitation Detection (AED) is to actively monitor and identify compromise of arbitrary, remote systems with the express intent to discover novel exploitation methods, track down elusive zero-day details, compile a list of known-compromised hosts, and most importantly get into the mind of today's cyber criminals. Simplistically, AED correlates changes visible to the remote monitoring system with external stimuli such as software patch schedules and security media sources in order to gain unique insight into the security threat landscape on an Internet scale. AED is a framework which is driven by arbitrary pluggable modules that must provide four high level implementations, namely port scanning, application identification via static and dynamic methods, and a data mining engine. The primary goal of this talk is to both present findings that trend the threat landscape of the Internet as a whole, and the tool itself, which is a means to introduce the audience to a number of best-of-breed open-source tools which have been integrated into this project.

2 Active Exploit Detection

The motivation for this project lies in the desire to increase the number of Internet systems under surveillance. Unlike many organizations, we do in fact have a large network of sensors that are monitoring the Internet both

in an inline capacity as well as through network span ports. While this is useful in its own right and currently scaled out to the degree which allows the results to be considered statistically viable, the chief limitation of this approach is that only traffic which crosses this sphere of inspection can be considered for analysis. Born out of this realization was the concept of an active monitoring system which could reach out and query an arbitrary host and could scale to the point that it could track the Internet as a whole. As a result of additional analysis on the threat landscape and prioritizing the goals of this project, we decided to focus on web applications and the lifecycle of exploit in this remarkably unique ecosystem.

2.1 Background and Previous Work

The first major requirement was a method to rapidly identify all the Internet hosts which are serving web applications of interest. The important thing here is speed, accuracy and the ability to integrate the results into an analysis engine. This requirement was split into two tasks, the first being a basic check to see if a web server is answering on a set of well-known HTTP ports and the second was to identify any common web applications that might be running on said host using a web application fingerprinting module. The goal here is to build a profile for each host which allows further filtering of the data to include only hosts which are running web applications that we wish to remotely monitor for exploitation. In other words, we need a way to limit the scope of the Internet to include web applications which are known at a given point in time to be vulnerable and as a result increase the odds of not only remotely detecting compromise but also gaining insight into what specifically happens to a machine post-compromise and how this informs the security threat landscape.

While still in the initial phases of development, we became overwhelmed with this looming sense of the massive amount of data storage that would be required to carry out this ambitious task. In order to accommodate this amount of data we found that a traditional relational database approach was inappropriate, due to the mixture of structured and unstructured data that did not easily lend itself to normalization, namely the web content itself, as well as the fact that the processing of this data did not realize the performance gains traditionally associated with indexing due to the commonality of table scans in the post-processing analysis routines. At this point we were also looking into a more robust analysis framework and discovered a striking

resemblance between what we were trying to do with AED and a number of very elegant solutions which had been worked out in meticulous detail for the problem of search engine design. The primary difference was that we must maintain more state in order to track changes and must do full text analysis on these changes in order to identify malicious modifications. However the benefits far outweighed the amount of customization we would have to do in order to accomplish the various tasks required by AED.

In the sections below, we will go into more detail on various open-source projects that we used to meet the requirements outlined above.

2.2 Port Scanning

When it came time to choose a network scanner, we initially began by writing our own based on a distributed TCP/IP stack written in Erlang. The reason that we started down this path was the desire to perform something called a "scatter connect" [18], although at the time we did not know or use this specific term. The idea is to have distributed cluster of machines which are logically grouped in threes, each of which participate in a separate stage of the TCP three way handshake. This stateless approach offers a number of speed improvements over traditional use of the underlying operating system network stack, given that we have a very specific and limited use case. The remote execution features of Erlang made implementation easier than it might seem upon first glance. Once we got to the testing phase, we stumbled upon *unicornscan* [18] which amazingly enough aligned with many of my requirements and methodologies and took things even a step further. Like our original design, TCP connection state tracking was moved out of kernel space using three separate user-land processes. The first process is a master process, which keeps track of which packets need to be sent, which process needs to send them, and correlates the responses. The second process is in charge of sending packets and the third process simply listens for responses. All of this comes to together to implement a basic user-land TCP/IP stack which is optimized for network scanning. At this point, we did some tests and found that not only did *unicornscan* outperform my attempt in Erlang, but it also bested *nmap* [19] considering the additional data processing to get the data into a consumable format for the analysis engine. The next step was straightforward, albeit not something that you would necessarily call simple: scan the entire Internet.

2.3 Application Identification

For web applications, specifically open source projects, there is a predictable correlation between specific point versions of an application and known exploits that have a very high success rate. This has been shown before and we believe this to be true. However, one of the questions that we set out to answer with AED was the approximate amount of time that a web application exploit is leveraged in the wild before it hits mainstream exploit databases and mailing lists. Furthermore, is it possible to detect more closely guarded exploits which have been used in conjunction with surgical attack campaigns? Our hypothesis was that by remotely monitoring web applications and correlating changes to hosts that were known at some point in the past to be running vulnerable web applications, that we should be able to predict updates to exploit databases and mailing lists and show cases where exploits are being retired and thus given away for "free" via these security media outlets. On the other side of this same exploit viability timeline, we should be able to trend "script kiddy" usage of low- to no-day exploits which are dropped on these same media outlets.

Thus the task of the web application identification module is to further filter the host list returned by the scanner module to those hosts which are running web application versions which we are interested in and then track changes to these versions over time on a per host basis. In evaluating many of the current tools which perform remote web application fingerprinting, there seems to be a major bifurcation in the approaches, which can be described as either dynamic or static. Static analysis relies of file presence alone to construct a specific point version fingerprint of a web application. At this time, the foremost example of a static analysis web application identification tool is *blindelephant* [25]. There are a number of strengths to this approach, namely that it is very fast and gives consistent results. The single largest downside to this approach that we discovered was its inability to account for small changes made to default web application installations, which seemed to be more common than we first thought.

The other option is a dynamic approach which inspects the entire content of a few carefully chosen pages. At first glance, this approach made a lot more sense in our application because we needed the rendered pages in order to perform the change tracking analysis. In this case, the use of static strings, program control flow and object oriented programming constructs proved to be a more efficient indicator of an application's version, given that these pages

were already fetched and available for analysis. Thus, the final point version calculation could be done offline without network access in a strongly parallel fashion which leveraged the advanced capabilities of the analysis framework we had chosen.

In the end we adopted a hybrid approach that did a rough web application family identification via static methods, then relied on dynamic analysis to narrow down the point version. In both approaches, a signature database must be constructed and maintained. This is a more complex operation in the dynamic case for sure, as the static method only needs to use the publically available source code tree to establish file existence. For the purposes of our application, we assume that these attributes are manually chosen and that the number of web application and version pairs will be kept to a manageable number. This is definitely an area for improvement and further recommendations to this end are included at the close of this paper.

2.4 Media Aggregation

The Media aggregation requirement simply asks for a method to be exposed which can be used to monitor and import data contained on specific security media outlets. Locations such as Zero Day Initiative [15], Security Focus [24], ExploitDB [11], Packet Storm [22], CERT [8] and NVD [21] provide a good start. It should be noted that in theory the entire crawl database that is used for change tracking could also be used as a media source, as part of a feedback loop. For example, a recent change in the number of less mainstream web sites that mention a particular web application or perhaps a code snippet that is known to be related to a web application could be a stimulus that could then be correlated back to a outward change on a monitored website. Along these lines, perhaps it could be argued that changes to a monitored website are newsworthy in and of themselves and you might wonder why we felt the need to correlate this back to a media event at all? Initially we added the media aggregation piece as a way to reduce the scope to a more manageable size via a time component, as web site changes are extremely common. However, as we soon found out the data explosion happened at the start of the pipeline, which makes sense in hindsight. We choose to keep this piece around for *ad hoc* queries and as we shall see it has remained very useful.

2.5 Change Tracking

In order to perform change tracking from afar, we needed to implement a crawler. The naive solution to this problem was quickly discovered to be inadequate. At this point two candidates were investigated which are believed to be best in class solutions. The first is *heritrix* [20] which is part of the Internet Archive project and *nutch* [7] which is a part of the Apache Software Foundation and has an intertwined fate with another Apache project discussed in the next section, namely *hadoop* [26]. Both projects are capable of Internet scale operation and as a result there is an inherent complexity that may be difficult to break through by the casual user. The advantage of *heritrix* is that the problem it solves, namely archiving and search, is closer to problem of change tracking than say *nutch* which is a true search engine implementation. Once the decision was made to build the analysis engine around *hadoop* it became apparent that the distributed file system *HDFS* [23] as well as the *MapReduce* [9] computational paradigm would be requirements as well. We will go into the details behind this decision in the next section. Suffice it to say, the only reason that *heritrix* was not removed from consideration immediately is that there has been work on a third-party *HDFS* writer [16] plug-in which we consider a necessary addition for this to be a viable option.

As we begin looking into these two projects, it became readily apparent that the designs are focused on first configuring the system and then running a command to carry out a series of steps in the fashion that the authors intended, rather than focusing on giving the end user the ability to develop a custom workflow that solves different or even closely related problems. In other words, both systems are monolithic rather than a loosely federated set of tools. Our application does not fall neatly into the problem space which is solved via the act of performing a vertical crawl to get specific content. Our needs are more along the lines of a web-mining approach, which requires the ability to do specialized post-processing of the data, which in the case of AED is either a comparison to a previous version or evaluation of a signature engine which detects suspicious or malicious content. To use search engine terms, our end result is different from that of a *lucene* [5] index or a list of inverted links.

Thus, we began work on a new crawler approach that would be more tailored to the requirements of AED which addresses the problems above and is compatible with the *hadoop* framework. During this process, we discovered

a project called *bixo* [17] which is very similar to what we started from scratch. Thus, we consider all three of these projects, namely *heritrix*, *nutch* and now *bixo* viable with the caveats given and are present and operational in the distributed virtual machine environment.

Despite adopting either the crawler or web-mining philosophy, the basic workflow is identical. A list of DNS entries, IP addresses, and URLs are injected into a database. The crawler selects work to be done from this list and normalizes this data and fetches the appropriate pages. The content is then saved and the work queue is updated appropriately. Then a pipeline which consists of content analyzers looks at the data and processes the data and stores it. Notice no mention of indexing, reverse linking, scoring or other operations typically associated with a search engine. We do maintain a feedback loop that allows us to crawl n levels deep from the initial seed list, something that we have not made extensive use of at this point. It is important to notice the pipeline nature of this entire process, something which will be exploited to scale this solution out.

2.6 Scalable Data Mining

Not surprisingly, if one attempts to tackle Internet-scale problems and researches novel solutions to these problems, at some point you will run across the work of the talented folks at Google. Aided by Google's seminal ideas of *MapReduce* [9] and the Google File System (GFS) [12], Yahoo! helped give birth to *hadoop* [26]. It is important to understand these two concepts as they are pivotal to the operation of *hadoop*. *MapReduce* [9] is a framework for processing massive datasets which decompose to distributable tasks that can be reduced to two steps: a map operation that transforms the input into an intermediate representation and a reduce function that recombines the input into the final output. The underlying data store used is Hadoop Distributed File System (HDFS) [23], which like GFS provides a fault-tolerant environment for working with very large files in a streaming data access model using inexpensive commodity hardware.

As you might expect, these concepts are almost a perfect fit for what AED is trying to accomplish. Initial findings showed a performance improvement over the relational database model that we were using originally. The reason is as follows. Massive updates are extremely efficient and small updates costly in this new paradigm, which is the inverse of a relational database. Thus the architecture set forth by *hadoop* shines in situations where full table

scans are the norm and true read write concurrency can be leveraged. In fact, comparisons between "redundant array of independent disks" (RAID) in a relational database context and a "just a bunch of disks" (JBOD) arrangement in a *hadoop* cluster have shown paradoxical speed improvements in the later. This is due to the fact that the current trend in disk drive performance shows that seek time is improving more slowly than transfer rate. Thus if our data access is dominated by seeks, it will take longer to read or write large portions of the dataset than streaming through it, which operates at or near the transfer rate. This can be applied to the problem of updates as well. Thus we can see that relational databases excel in the areas of point queries and updates, while the *MapReduce* strategy excels in cases where we need to explore the entirety of the dataset in a batch fashion. Finally, the other difference lies in the ability to handle a mixture of structured and unstructured data. Databases excel at handling formally structured data, but fall short in the semi-structured case where there might be a schema, but that schema is often ignored. This is extremely common in the world of web application protocols. The ability of *MapReduce* to interpret the data at processing time so to speak which is informed by a *ad hoc* set of input keys and values for *MapReduce* is something that we found to be a huge strength to the *MapReduce* approach.

While *MapReduce* is at the core of *hadoop*, it is by no means the only game in town. In fact, we leverage a number of other members of the *hadoop* ecosystem. *Hive* [4] is a distributed data warehouse, which manages data in *HDFS* and provides a SQL-like query language which is translated by the runtime engine to *MapReduce* jobs. Another important player behind the scenes is *hbase* [3], which is a distributed, column-oriented database which uses *HDFS* for its underlying storage and allows for random reads. Finally, we use *sqoop* [14] which is a tool that allows us to efficiently move data between relational databases and *HDFS*. As you might guess, this is the bridge that we use to integrate the PostgreSQL instance which sits on the backend of *unicornscan* [18] and leverages *hbase* along with *hive* to easily manipulate the imported data.

As a side note, the *hadoop* framework is also naturally suited to log analysis and processing, a use case we are exploring in parallel with this project to aid with processing of massive amounts of sensor logs.

3 Results

The scope of this problem is vast. Amortized actuals show that we need around 1 terabyte to store 100 million pages which can be crawled by a single machine whose specifications include 1 CPU and 1 gigabyte RAM. Our scan results have shown that there are approximately 104 million active hosts, of which 9.2 percent can be identified as running a web application. If we assume around 10 kilobytes per page and a monthly refresh time, along with a rough number of 1 billion pages per month, we come up with a bandwidth requirement of 40 megabits per second inbound and ten crawlers who are also hosting a piece of the distributed file system which totals around 10 terabytes. Thus, we selected a set of web applications that would reduce these numbers by a factor of 10. At this point it should be noted that cloud services such as Amazon Elastic Compute Cloud (EC2) [1] could be considered an excellent viable option to the more brute force approach of acquiring and maintaining a rack of servers ourselves and something we are currently investigating. The idea would be to use Amazon S3 [2] as a data store and then run the cluster on EC2 by means of Apache Whirr [6], which is a set of scripts which automates the running *hadoop* on EC2 and other cloud providers.

As the last few programs continue to run and we begin generating graphs and snippets from compromised websites, we regret to say that this information will be found in the accompanying presentation slide deck and this whitepaper will be updated within the prescribed time window after the conference to also include that information here.

4 Future Work

In the immediate timeframe, we feel that it would be nice to leverage the modular approach that *unicornscan* [18] offers to do some basic passive web application fingerprinting. This particular function we feel could be carried out by *unicornscan* natively and this eliminates the need for the initial passive probe via the web application fingerprinting module. As part of our hybrid approach, further active fingerprinting would then be done offline once the crawled pages have been stored in *HDFS*. We feel that there are significant advantages to this approach. The long term solution would be to in fact rewrite *unicornscan* in a fashion that would allow it to run directly on the

hadoop cluster. Thus would be an interesting project in its own right and we would expect a performance gain as a result of this tighter integration which would not only eliminate the *sqoop* [14] bridge, but also give us the ability to run this module on a borrowed cloud infrastructure.

Another relatively minor issue which merits investigation since it affects the speed in which we can crawl the Internet is the fact that many of the crawl libraries use standard synchronous Java IO. The problem with this is that for a given number of documents that we wish to crawl, we must have that same number of threads active. This becomes costly when we begin to fetch on a larger scale and any optimizations made to each thread are negated by the overhead of the threading library itself. By moving to an asynchronous approach we expect a drastic improvement in crawl throughput.

Somewhat unrelated to the problem at hand, but nevertheless a very interesting problem that we would like to solve deals with the fact that many malicious sites, specifically malware depots, are using browser detection to not only avoid automated systems such as AED, but also deliver an appropriate browser-based attack to an end user, which is tailored to the specific browser type and version that the potential victim is using. This use case which resists the native crawler libraries present in the various frameworks which we have investigated, in addition to our inability to monitor legitimate websites which are leveraging Ajax-driven Web 2.0 applications which require execution of JavaScript in order to discover obfuscated and dynamically generated links means that we need to use a more sophisticated browser head, such as what is offered by *HTMLunit* [13] or even in the extreme cases, a full featured browser such as Internet Explorer or Firefox.

Further out, we are looking for ways to investigate and crawl the "deep-net" [10]. Despite any feedback mechanism which we have investigated so far, the nature of the problem is that we only will naturally discover a relatively small portion, yet still into the billions, of web pages. This in fact is an interesting research problem in the web crawler space that we think merits more investigation, as this is precisely the type of places we expect attackers to be hiding. Again, this is not directly related to the functions required by AED which target specific web applications and is not concerned with the links on these sites to the rest of the Internet, but nonetheless something we would like to be able to handle so that we could begin to use the same framework to monitor known malicious websites.

References

[1] *Amazon Elastic Compute Cloud*. URL: http://aws.amazon.com/ec2.

[2] *Amazon Simple Storage Service*. URL: http://aws.amazon.com/s3.

[3] *Apache HBase*. URL: http://hbase.apache.org.

[4] *Apache Hive*. URL: http://hive.apache.org.

[5] *Apache Lucene*. URL: http://lucene.apache.org.

[6] *Apache Whirr*. URL: http://incubator.apache.org/whirr.

[7] Mike Cafarella and Doug Cuttin. *Building Nutch: Open Source Search*. Apr. 2004. URL: http://queue.acm.com/detail.cfm?id=988408.

[8] *Carnegie Mellon University's Computer Emergency Response Team*. URL: http://www.cert.org/cert/.

[9] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. June 2009. URL: http://labs.google.com/papers/mapreduce.html.

[10] *Deep Web*. URL: http://en.wikipedia.org/wiki/Deep_Web.

[11] *Exploit Database*. URL: http://www.exploit-db.com.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google File System*. Oct. 2003. URL: http://labs.google.com/papers/gfs.html.

[13] Marc Guillemot and Daniel Gredler. *HtmlUnit: An Efficient Approach to Testing Web Applications*. Jan. 2009. URL: http://javasymposium.techtarget.com/html/images/MGuillemot_HtmlUnit.pdf.

[14] *Hadoop Sqoop*. URL: https://github.com/cloudera/sqoop/wiki.

[15] *HP TippingPoint's Zero Day Initiative*. URL: http://www.zerodayinitiative.com.

[16] Doug Judd. *Heritrix Hadoop DFS Writer Processor*. Jan. 2007. URL: http://corporate.zvents.com/developers/thelab.html.

[17] Doug Judd and Stefan Groschupf. *Bixo - A Webcrawler Toolkit*. May 2009. URL: http://bixo.101tec.com/wp-content/uploads/2009/05/bixo-intro.pdf.

[18] Robert Lee and Jack Louis. *Introducing Unicornscan: Riding the Unicorn*. July 2005. URL: https://www.defcon.org/images/defcon-13/dc13-presentations/DC_13-Lee.pdf.

[19] Gordon "Fyodor" Lyon. *Nmap: Scanning the Internet*. Aug. 2008. URL: https://www.blackhat.com/presentations/bh-usa-08/Vaskovich/BH_US_08_Vaskovich_Nmap_Scanning_the_Internet.pdf.

[20] Gordon Mohr et al. *An Introduction to Heritrix: An Open Source Archival Quality Web Crawler*. Sept. 2004. URL: http://iawaw.europarchive.org/04/Mohr.pdf.

[21] *National Vulnerability Database*. URL: http://nvd.nist.gov/.

[22] *Packet Storm*. URL: http://packetstormsecurity.org/files.

[23] Konstantin Shvachko et al. *The Hadoop Distributed File System*. May 2010. URL: http://storageconference.org/2010/Papers/MSST/Shvachko.pdf.

[24] *Symantec SecurityFocus/Bugtraq*. URL: http://www.securityfocus.com/vulnerabilities.

[25] Patrick Thomas. *Blind Elephant: Web Application Fingerprinting and Vulnerability Inferencing*. July 2010. URL: https://media.blackhat.com/bh-us-10/presentations/Thomas/BlackHat-USA-2010-Thomas-BlindElephant-WebApp-Fingerprinting-slides.pdf.

[26] Tom White. *Hadoop: The Definitive Guide, Second Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2010.