Heat-seeking honeypots: design and experience

John P. John
University of Washington
Seattle, WA
jjohn@cs.washington.edu

Fang Yu
Microsoft Research
Silicon Valley
fangyu@microsoft.com

Yinglian Xie
Microsoft Research
Silicon Valley
yxie@microsoft.com

Arvind Krishnamurthy
University of Washington
Seattle, WA
arvind@cs.washington.edu

Martín Abadi
Microsoft Research
Silicon Valley
abadi@microsoft.com

ABSTRACT

Many malicious activities on the Web today make use of compromised Web servers, because these servers often have high pageranks and provide free resources. Attackers are therefore constantly searching for vulnerable servers. In this work, we aim to understand how attackers find, compromise, and misuse vulnerable servers. Specifically, we present *heat-seeking honeypots* that actively attract attackers, dynamically generate and deploy honeypot pages, then analyze logs to identify attack patterns.

Over a period of three months, our deployed honeypots, despite their obscure location on a university network, attracted more than 44,000 attacker visits from close to 6,000 distinct IP addresses. By analyzing these visits, we characterize attacker behavior and develop simple techniques to identify attack traffic. Applying these techniques to more than 100 regular Web servers as an example, we identified malicious queries in almost all of their logs.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security, Web applications

1. INTRODUCTION

Compromised Web servers are commonly utilized for conducting various nefarious activities such as serving phishing and malware pages, acting as open proxies, and redirecting user traffic to malicious sites. Recent reports suggest that almost 90% of Web attacks take place through legitimate sites that have been compromised [4]. In addition, attackers populate compromised servers with content containing popular search keywords in order to poison search results from search engines. Over 50% of these keywords have at least one malicious link to a compromised site in the top search page [7].

While it could be argued that attackers could always set up their own Web servers, compromised Web servers are highly desirable as they often have high pageranks. Hence, anything hosted on these servers would be more trusted and more likely to attract traffic than content hosted on a new server. Furthermore, compromised Web servers can communicate with a large number of client machines that are behind NATs and firewalls, which would otherwise be hard for attackers to reach directly. Finally, attackers are able to utilize these resources for free while simultaneously reducing the likelihood of being tracked down by security analysts.

Understanding how attackers identify Web servers running vulnerable applications, how they compromise them, and what subsequent actions they perform on these servers, would thus be of great value. A natural approach to studying attack patterns and attacker behavior in this setting would be to make use of honeypots. Specifically, we want honeypots that can emulate Web applications; these honeypots will be run on Web servers, and so are a form of server-based honeypots. Unlike client-based honeypots, which can initiate interactions with attackers (by visiting suspicious servers, executing malicious binaries, etc.), server-based honeypots are passive, and have to wait for attackers. The biggest challenge for us, therefore, is how to effectively get attackers to target our honeypots. The next challenge is how to select which Web applications to emulate. There are a large number of vulnerable applications available, and actually installing and running all of them would be time-consuming, human-intensive, and therefore not scalable.

To address these challenges, we design heat-seeking Web server-based honeypots. As the name suggests, we actively adapt our honeypots to emulate the most popular exploits and construct pages that are similar to the ones targeted by attackers. For this purpose, our heat-seeking honeypots have four components. First, they have a module to identify Web pages that are currently targeted by attackers. Previous work has shown that many attackers find vulnerable servers with the help of Web searches [14, 17]. Therefore, we look through the search logs of the Bing search engine [1] to identify queries issued by attackers. Second, we generate Web pages based on these queries without manually setting up the actual software that is targeted. We then advertise these links through search engines, and when we receive attack traffic we log all interactions with the attackers. Finally, when analyzing the honeypot logs, we design methods to separate attacker traffic from crawler traffic and legitimate user traffic.

Our heat-seeking honeypots have the following attractive properties:

1. Honeypot pages can be generated automatically without understanding the latest exploits or knowing which software is affected.

2. Web pages are put up to attract attackers, without the heavy overhead of installing the real software. After analyzing attacker visits, if we need additional information about a particular attack, we can set up a high-interaction honeypot by installing the actual software.

3. Since the honeypot pages look identical to pages generated by actual software, the attackers assume the presence of the software and conduct attacks. This allows us to observe a wide range of attacks.

We implement and deploy our heat-seeking honeypot under a personal home page on the `cs.washington.edu` domain. Despite its relatively obscure location, during the three months of operation, it attracted more than 31,000 attacker visits from 6,000 distinct IP address. When compared to a setup where real vulnerable applications were installed, our honeypot pages see a similar amount of traffic, albeit at a much lower setup cost. We observe a wide variety of attacks including password guesses, software installation attempts, SQL-injection, remote file inclusion attacks, and cross-site scripting (XSS) attacks.

By analyzing the attack traffic observed at our honeypots, we characterize attacker behavior and general trends in Web attacks. We also develop simple techniques to distinguish legitimate requests from attack requests. Applying this technique to a set of random Web servers on the Internet, we identify malicious accesses in all of their logs, sometimes accounting for 90% of the entire traffic.

The rest of the paper is structured as follows. We discuss related work in Section 2. In Section 3, we describe the design and implementation of our honeypot. We then present the results of our honeypot deployment in Section 4. We discuss ideas for increasing the effectiveness of honeypots and securing Web applications in Section 5 and conclude in Section 6.

2. RELATED WORK

Honeypots are usually deployed with the intent of capturing interactions with unsuspecting adversaries. The captured interactions allow researchers to understand the patterns and behavior of attackers. For example, honeypots have been used to automate the generation of new signatures for network intrusion detection systems [13, 20], collect malicious binaries for analysis [9], and quantify malicious behavior through measurement studies [15].

Client-based honeypots are used to detect malicious servers that attempt to exploit clients. They are typically used to find sites that try to exploit browser vulnerabilities, or perform drive-by downloads [15, 21]. Server-based honeypots such as ours, on the other hand, emulate vulnerable services or software, and wait for attackers.

Further, depending on the level of interaction with the adversary, honeypots can be categorized as low-interaction or high-interaction. Low-interaction honeypots such as Honeyd [16] emulate some specific portion of the vulnerable service or software. These honeypots function only when the incoming requests match specific known patterns. High-interaction honeypots, on the other hand, can handle all requests, and they reply with authentic responses through the use of sandboxed virtual machines running the actual software [10].

Several Web honeypots have been developed to capture attack traffic directed at vulnerable Web applications. The Google Hack Honeypot [8] contains around a dozen templates for common Web applications, including a high interaction php-based shell. Other recent honeypots [2, 3] add additional templates and also include a central server where all honeypot logs are collected. They also save copies of files which attackers might try to upload, for later analysis. However, these Web honeypots emulate a limited number of vulnerabilities and require manual support, thereby not allowing them to scale easily.

The honeypot by Small et al. [18] uses natural language processing to generate responses to attacker requests, based on a training set containing vulnerable Web pages. The heat-seeking honeypots are simpler and work with any software without training. We also capture a wider class of attacks, which include attacks on software discovered by attackers through the specific directory structure used by the software.

3. SYSTEM DESIGN

We want heat-seeking honeypots to attract many attackers in order to study their behavior. We also want our honeypots to be generic across different types of vulnerable software and to operate as automatically as possible—from generation of Web pages to detection of attack traffic. Human experts are needed only to do post-analysis of detected attack traffic.

In our design, the heat-seeking honeypots have four components. The first component is to identify which types of Web services the attackers are actively targeting. The second component is to automatically set up Web pages that match attackers' interests. The third component advertises honeypot pages to the attackers. When the honeypot receives traffic from attackers, it uses a sandboxed environment to log all accesses. Finally, we develop methods to distinguish attacks from normal users/crawler visits and perform attack study.

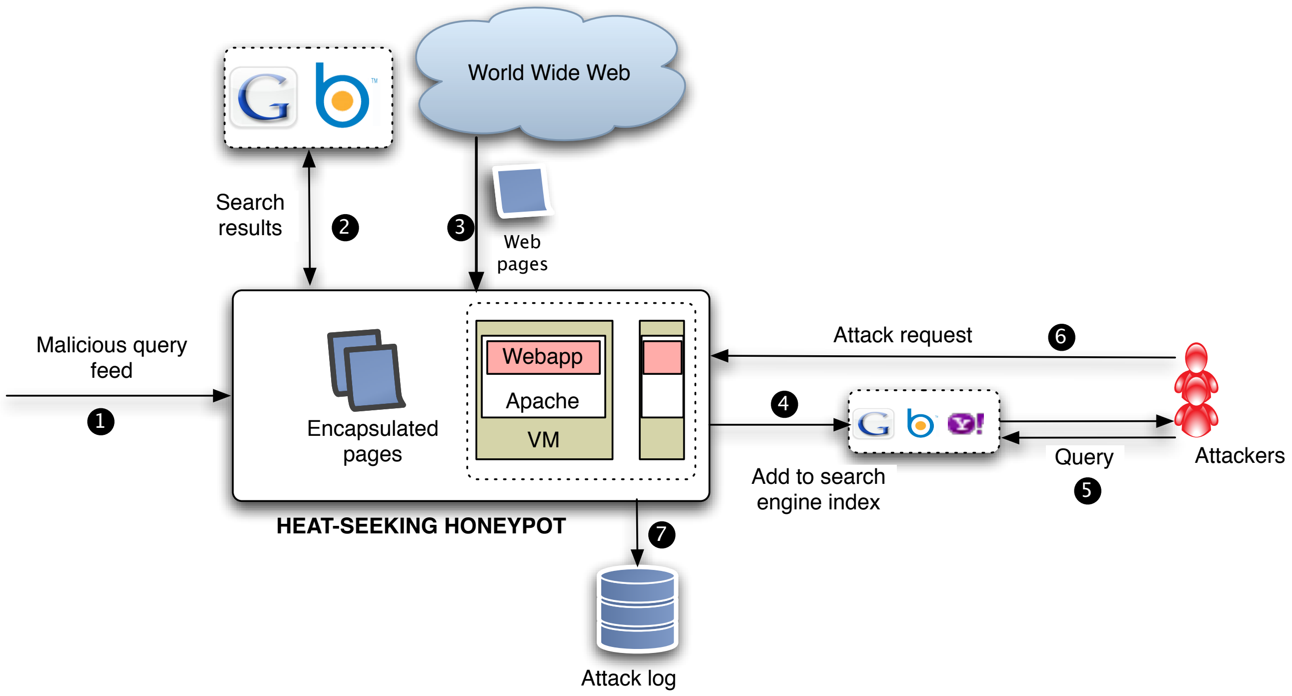

Figure 1: The working of our heat-seeking honeypot. Attacker queries from feed (1) are issued to search engines (2). The pages from the search results are fetched (3). These pages are encapsulated and put up on the heat-seeking honeypot, along with other real software installed on VMs. Then these pages are advertised and crawled by search engines (4). When attackers issue similar queries to search engines, the honeypot pages are returned in the search results (5). When attackers interact with the honeypot (6), all interactions are logged (7) for further analysis.

In the rest of this section, we explain these components in detail. Some components can be implemented in different ways. We discuss design tradeoffs and then explain the design choices we made in our prototype system.

3.1 Obtaining attackers' queries

Attackers often use two methods to find vulnerable Web servers. The first is to perform brute-force port scanning on the Internet. The second is to leverage search engines. It is reported that vulnerable Web servers are compromised immediately after a Web search reveals their underlying software [14]. For example, a query `phpizabi v0.848b c1 hfp1` to a search engine will return to the attacker a list of Web sites that have a known PHP vulnerability [11]. Because of the convenience of this approach, many attackers adopt it, and our system is designed to attract such attackers.

We use SBotMiner [22] and SearchAudit [12] to automatically identify malicious queries from attackers in the Bing log. SbotMiner combines attack queries into groups and generates regular expressions. Each group is likely to have been generated by the same attacker, typically using the same script [22]. An example of a query group is
`inurl:/includes/joomla.php [a-z]{3,7}`
Here, the attacker is searching for sites where the URL contains a particular string. The second part of the regular expression corresponds to random English words added by the attacker to diversify the query results. From each such group, we pick the most popular query by the attacker and feed it to the next component.

3.2 Creation of honeypot pages

Given the query used by the attacker to find a set of servers, how do we create an appropriate honeypot? We explore a few different approaches and describe their pros and cons.

(a) **Install vulnerable Web software:** This approach sees exactly how the attacker interacts with and compromises the software, but it it requires a domain expert to manually identify the target software and set up the software.

(b) **Set up Web pages matching the queries:** Instead of setting up the actual software, we can create Web pages that are similar to the ones created by the software. By issuing the query ourselves to different search engines, and looking at the Web pages returned in the results, we can generate pages which match the attacker's criteria. The advantage here is that the whole process can be automated, thus allowing it to scale. The disadvantage with this approach is that it leads to fewer interactions with the attacker. Since there is no real software running, requests made by the attacker may result in incorrect responses, thereby limiting the depth of the interaction.

(c) **Set up proxy pages:** This approach is similar to the previous one, but instead of creating a copy of the Web pages in the search results, we set up a transparent proxy, which forwards all requests to the actual Websites. This proxy setup can be automated, allowing the approach to scale. While this approach combines the advantages of the previous two approaches, it has one major drawback — we will be forwarding to vulnerable servers malicious traffic that may potentially compromise the servers, thereby assisting in malicious attacks.

In our deployment, we chose a combination of options (a) and (b), and did not adopt option (c).

For each query obtained in the previous component, we issue it to the Bing and Google search engines, and collect the result URLs. We pick the top three results as the ones we wish to emulate. Our crawler fetches the Web pages at these URLs, along with the other elements required to render these pages (such as images and css stylesheets).

To ensure that visiting a page will not cause malicious traffic to be sent to other sites, we strip all Javascript content from the page and rewrite all the links on the page to point to the local versions. The HTML content within the page is encapsulated in a PHP script, which simply returns the HTML, and logs all information regarding each visit to a database. Also, we mirror the same URL structure in our honeypot pages as in the original, so as to match queries where the attackers specify the URL structure. Considering again our earlier example, the honeypot page would be stored at:
`http://path/to/honeypot/includes/joomla.php`
Any of the random keywords used by the attacker in the queries (as obtained from our query groups) are also sprinkled across the honeypot page in order to increase the chance of a match.

We also manually install a few common Web applications that were frequently targeted, learned from our honeypot pages. These applications are installed in separate VMs, so that if one of them gets compromised, it doesn't affect the working of the other applications. We use virtual hosts and port forwarding to allow access to the Web servers running on the VM. In order to check if an application has been successfully compromised, we monitor the filesystem to check if any of the application files have been modified, or if new files have been added.

3.3 Advertising honeypot pages to attackers

Now that we have setup honeypot pages that attackers are interested in, our next task is to effectively advertise our pages.

Ideally, we want our honeypot pages to appear in the top results of all malicious searches, but not normal users' searches. But this requires the help of all major search engines to detect attack traffic in realtime and deliver our honeypot pages when they detect malicious searches.

In our deployment, we simply submit the URLs of the honeypot pages to the search engines and then boost the chance of honeypot pages being delivered in response to malicious queries by adding surreptitious links pointing to our honeypot page on other public Web pages (such as the homepages of the authors). The links are added in such a way that they are not prominently visible to regular users. This reduces the chance of having regular users randomly reach the pages, while still allowing search engines to crawl and index these links. Having these additional links pointing to the honeypot increases the pagerank of the honeypot pages, and therefore increases the likelihood of having our honeypot URLs returned to the attackers.

3.4 Detecting malicious traffic

We log all visits to our heat-seeking honeypots. Next, we need to process the log to automatically extract attack traffic.

Different from traditional honeypots that operate on darknets or locations where no legitimate traffic is expected, our honeypot is likely to receive two kinds of legitimate traffic. First, we expect search engine crawlers to visit our pages since the pages are publicly accessible. Second, once the pages are indexed by search engines, it is possible (though unlikely) for regular users to stumble upon our honeypot.

3.4.1 Identifying crawlers

One might imagine it is easy to identify crawler traffic by looking for known user agent strings used by crawlers. For example, Google's crawler uses `Mozilla/5.0 (compatible; Googlebot/2.1;+http://www.google.com/bot.html)` as its user agent. However, this approach does not always work. As the user agent string can be trivially spoofed, attackers can also use a well-known string to escape detection. In fact, we observed several instances of this phenomenon in our experiments. In addition, there are many user agent strings used by different crawlers. It is hard to enumerate all of them, especially for small or new crawlers.

In our work, we identify crawlers by their access patterns—crawlers typically follow links in the page and access most of the connected pages. We first characterize the behavior of a few known crawlers and then use such knowledge to detect other crawlers.

Characterizing the behavior of known crawlers

We take a few known crawlers by looking at the user agent string and verify that the IP address of the crawler matches the organization it claims to be from. Note that a single search engine can use multiple IP addresses to crawl different pages on a Web site. Therefore, we first map IP addresses to Autonomous Systems (AS) owned by the search engine and look at the aggregate behavior. Since these are legitimate crawlers, we use their behavior to define the baseline.

There are two types of links a crawler would visit: static and dynamic links. Those automatically generated honeypot pages are static ones. Dynamic links are generated by the real software, where URL arguments can take different values. For example, a page allows a user to register for an account has the form:
`/ucp.php?mode=register&sid=1f23...e51a1b`
where sid is the session ID, which changes with each request.

To distinguish static links from dynamic ones, we look at histograms of URLs accessed by known crawlers. For those that are accessed by only one crawler, we treat them as dynamic link candidates.

Despite the uniqueness of dynamic links, the structure of URLs and variables is well formed and hence they can be expressed using regular expressions. We feed the dynamic link candidates to the AutoRE [19] system for generating regular expressions. This system allows us to automatically generate a set of patterns E, describing the eligible values for each variable in different URLs. For our previous example, AutoRE gives us the pattern
`/ucp.php mode=register sid=[0-9a-f]{32}`.

After detecting all dynamic links, we treat the rest as static links and we record their union to be the set of crawlable pages C.

Identifying unknown crawlers
We identify other IP addresses (also grouped by AS numbers) that have similar behavior as the known crawlers. Here "similar" is defined in two parts. First, they need to access a large fraction of pages P in the crawlable pages C. In our work, we use a threshold $K = |P|/|C|$ for selecting crawlers. This threshold is easy to set as most crawlers accessed a vast majority of pages (detailed in Section 4.1.1). Second, outside of the set of pages in C, crawlers should access only pages that match regular expressions in E.

3.4.2 Identifying malicious traffic

The static Web pages on the heat-seeking honeypot are just used to attract attackers' visits. From our honeypot logs, we observed that most of the attacks are not attacking these static pages. Rather, they try to access non-existent files, files that were not meant to be publicly accessed, or providing spurious arguments to scripts, assuming that the full software package is installed and running. For static Web pages on the honeypot, since we know the exact set of links that are present in our honeypot, we can treat this set as our *whitelist*.

There are also a few links that needed to be added to the whitelist. Most well-behaved crawlers look for `robots.txt` before crawling the site. For sites that do not have `robots.txt`, crawler's request violates the whitelist. Another similar type of access to non-existent files comes from client browsers. Browsers may request `favicon.ico` when they visit the site. This file contains a small icon for the site, which is displayed by the browser, but not all Web servers have it. Such links should be added to our whitelist.

Requests for links not in the whitelist are considered suspicious. This approach is different from frequently used blacklist-based techniques, where illegal access patterns are explicitly constructed by human operators or security experts. In contrast, whitelist-based approaches can be automated and thus more generally applied to different types of software.

This whitelist-based approach can be generalized to identify malicious traffic for regular Web servers on the Internet. Each site master could enumerate the list of URLs from the sites and put them into the whitelist. Alternatively, one could also use the set of links accessed by a known crawler

Crawler	Number of IPs	Number of pages crawled
Google	109	156
Bing	206	147
Yahoo	20	127

Table 1: The the number of IP addresses used by the three big search engines, and the number of pages crawled.

(C) as the whitelist, provided every link on the site is accessible [1] to the crawler. For dynamic links, one can generate regular expressions based on normal user accesses, or site masters can handcraft those. In Section 4.4, we present results on applying this approach to 100 random Web servers without manual effort.

4. RESULTS

In this section, we present results from a prototype deployment of our system. We set up heat-seeking honeypots under a personal home page at the University of Washington. During three months of operation, our system consisted of 96 automatically generated honeypot Web pages (3 pages each from 32 malicious queries), and four manually installed Web application software packages. They altogether received 54,477 visits from 6,438 distinct IP addresses.

We first present results on how we distinguish malicious visits from legitimate ones. Then we characterize the types of malicious requests received at the honeypots. In addition, we compare different honeypot setups and demonstrate the effectiveness of heat-seeking honeypots in terms of attracting malicious traffic. Finally, we apply the whitelist approach learned from operating the honeypots to other Web-logs and show its effectiveness in identifying malicious requests.

4.1 Distinguishing malicious visits

4.1.1 Crawler visits

To filter traffic from crawlers, we first map IP addresses to ASes as mentioned in Section 3.4. Then we pick three popular search engine crawlers, Google, Bing and Yahoo, and mark the URLs they visit. Not all crawlers visit all the pages set up by our system; this is probably because our honeypot pages have a relatively low pagerank, so are not the highest priority for crawlers. Table 1 shows the number of links visited by each of the major crawlers.

Our Web honeypot, consisting of only static HTML pages, does not contain any dynamic links. The software honeypot, however, has many dynamic links and we look at how well our approach of detecting dynamic links works. For each link that is visited by search engine crawlers, we count the number of different crawlers that crawled the link. The idea here is that a static link remains same across sessions, so all the crawlers will have crawled the same link. A dynamic link, on the other hand, changes in each session. So, each crawler will see a slightly different link – therefore each dynamic link will be crawled by exactly one crawler.

[1]For sites that specify restrictions in `robots.txt`, the disallowed links should be added to the whitelist.

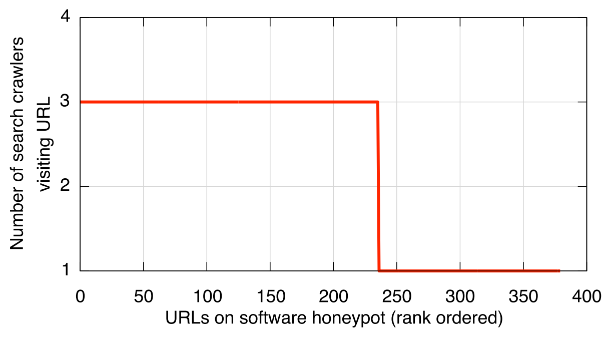

Figure 2: The number of search engines visiting each URL in the software honeypot.

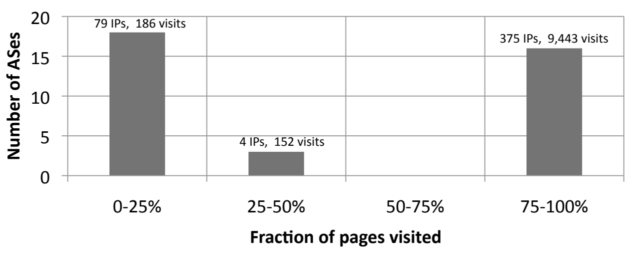

Figure 3: The number of legitimate visits from distinct ASes for different number of total pages visited.

Figure 2 shows the number of crawlers accessing each link. Around half of the links in the software honeypot see all three of the big crawlers visiting, but the other half see exactly one crawler. We manually checked these, and unsurprisingly, these correspond to the dynamic links in the software. Therefore, the simple approach of counting the number of crawler visits helps us identify potential dynamic links in the Web application.

The dynamic links are fed to AutoRE, which generates a total of 16 regular expressions - these determine the values allowed for each variable in the dynamic link. Some examples of the kinds of regular expressions generated for these dynamic links are:

```
/oscommerce/shopping_cart.php osCsid=[0-9a-f]{32}
/phpBB3/style.php sid=[0-9a-f]{32}
```

After getting the list of crawler-visited URLs, we examine our logs to identify other IP addresses that visit URLs that are a strict subset of the crawlable static URLs or match the regular expressions for the dynamic URLs. This represents the set of visitors that did not perform any malicious activity – they are either crawlers or legitimate visitors. For each AS, we compute the fraction of crawlable pages visited. The distribution is plotted in Figure 3. We find this distribution to be bi-modal, with 16 ASes (comprising 375 different IP addresses) crawling more than 75% of the hosted pages, and another 18 ASes (comprising 79 IP addresses) visiting less than 25% of the pages. We therefore pick 75% as the threshold K, and anyone visiting more than this is considered a crawler, while others are considered legitimate users that reach our honeypot pages.

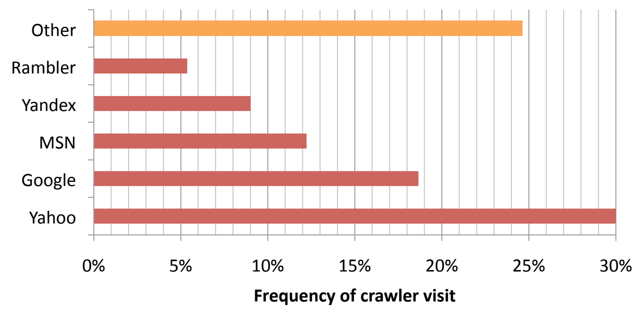

Figure 4: The frequency with which each crawler visits us.

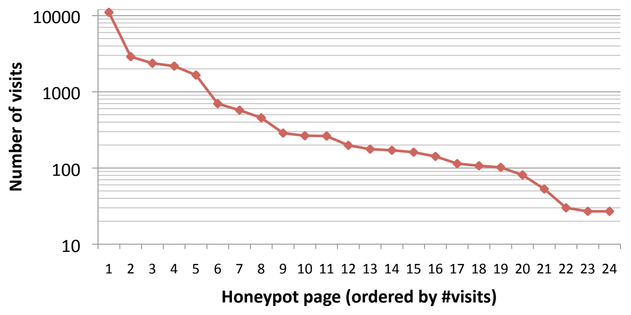

Figure 5: The number of attacker visits to each honeypot page. We only show pages with 25 or more visits.

In all, we found and verified 16 different crawlers. The 375 crawler IP addresses account for 9443 visits. Figure 4 shows how often each crawler visits our honeypot pages. We see that Yahoo, Google, and Bing are the three most frequent crawlers, followed by Russian sites Yandex and Rambler. Nearly a quarter of the visits were from other crawlers which were mostly specialized search engines (such as `archive.org` which tracks older version of Web pages).

4.1.2 *Attacker visits*

After eliminating accesses from crawlers, there are 79 IP addresses that access only whitelisted URLs. These could either be normal users who visited our pages accidentally, or from attackers who did not launch any attacks. We conservatively consider these as potential attack visits and count them as false negatives for our malicious traffic detection approach. The remaining traffic is malicious, trying to access URLs that are not in the whitelist. We find 5,980 attacker IP addresses responsible for 44,696 visits (we classify these attacks into various categories in Section 4.2.3). Thus, the false negative rate of our approach is at most 1%.

In Figure 5, we plot the total number of visits by attackers to each honeypot page. This gives us a measure of which of our pages is most effective at drawing attackers. Out of the 96 honeypot pages we set up, we only plot those pages which received at least 25 visits, after excluding crawler visits. As we can see from the figure, different page received drastically different number of attacks. The most popular honeypot page with over 10,000 visits was for a site running *Joomla*, a content management system. The differences in popularity can be caused by the frequency of attacks, and

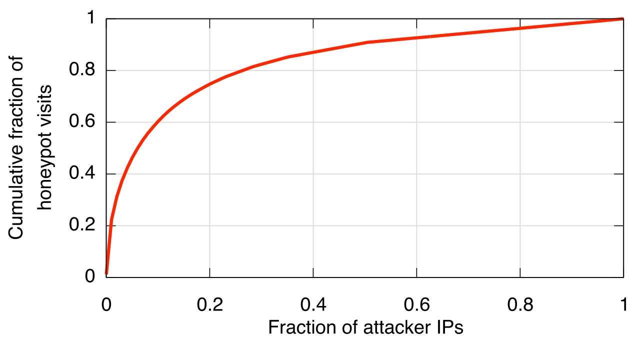

Figure 6: The cumulative distribution of attacker IP addresses, and the number of honeypot pages visited.

Country	% of IPs
United States	24.15
Germany	5.94
China	5.34
Russia	4.75
France	4.31
England	3.68
Poland	3.62
South Korea	3.48
Indonesia	3.27
Romania	2.86

Table 2: The top 10 countries where the attacker IP addresses are located.

also by the pageranks of other vulnerable Web servers in the search results. If attackers search for queries that return mostly low-ranked pages, the chance of our honeypot being returned in the top results is much higher.

4.2 Properties of attacker visits

4.2.1 *Attacker IP properties*

We look at the behavior of attacker IP addersses when visiting our honeypot pages, in particular the number of attempts by each attacker. From Figure 6, we observe that 10% of the IP addresses are very aggressive, and account for over half of the page visits. These aggressive IP addresses are the ones which make multiple requests (usually several hundred) to the same page, trying different arguments in order to find vulnerabilities.

We also look at the geographic locations of the attacker IP addresses, and find that over 24% of the IP addresses are located in the United States. Table 2 shows the top ten countries from where the honeypot visits originate.

4.2.2 *Discovery time*

Here, we try to answer the question: how soon after a page is setup does it start getting hit by attackers? We calculate the number of days between the first crawl of the page by a search crawler and the first visit to the page by an attacker. We call this the *discovery time*, i.e., the time it takes for the attackers to find the page after it has been indexed. Figure 7 shows the distribution of discovery times for the various honeypot pages. For two of the honeypot pages, we found attacks starting the same day as it was crawled. Because those pages were for a not very popular search term, so even though the pages are new, they show up high in search results, and are targeted by attackers. In the median case, however, we find the discovery time to be around 12 days.

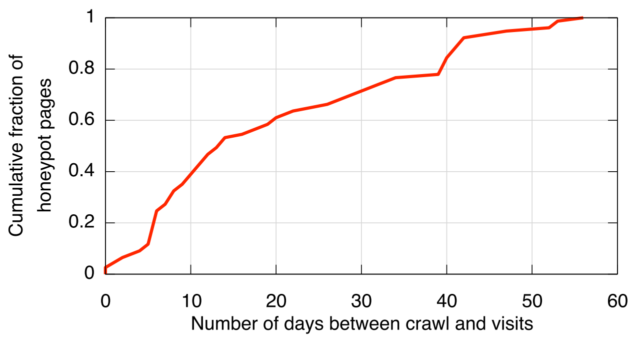

Figure 7: The distribution of the number of days between the honeypot page being crawled and being attacked.

4.2.3 *Attacks seen*

We classify all the attacker requests we see into ten different categories, based on the attacker's motives. The categories are created manually, and are not intended to be a complete classification of all possible attacks – their purpose is to provide an overall idea of what the attackers are trying to do. A brief explanation of each attack category, an associated example, and the fraction of traffic observed at the honeypot is shown in Table 3.

The most common attack we see on our honeypot is from attackers looking for file disclosure vulnerabilities (FILE). Here, attackers try to access files on the file system which are not meant to be accessed. This kind of attack happens when Web applications do not correctly implement access control, and anyone is allowed to access files on the filesystem with the privilege of the Web server. Examples of this typically include attempts to access the `/etc/passwd` file on UNIX machines, application configuration files containing database credentials, etc. We also see a lot of attackers trying to find a particluar `xmlrpc.php` (XMLRPC) file with a known vulnerability. Several popular applications shipped with a vulnerable version of this file a few years ago, and though it was fixed subsequently, attackers still look for it. On a similar note, attackers also look at several other PHP files to see if the allow for remote file inclusion (RFI). This bug allows the application script (typically PHP) to execute code included from another file. Sometimes the file can be in a remote location, and this allows an attacker to execute arbitrary code on the Web server.

A surprising find in the table is that the two most commonly heard-of vulnerabilities, cross-site scripting (XSS) and SQL injection (SQLI), are not very popular in our honeypots. A possible explanation for this is that these attacks are not tar-

Category	Description	Example	Traffic (%)
ADMIN	Find administrator console	`GET,POST /store/admin/login.php`	1.00
COMMENT	Post spam in comment or forum	`POST /forum/reply.php?do=newreply&t=12`	
FILE	Access files on filesystem	`GET /cgi-bin/img.pl?f=../etc/passwd`	43.57
INSTALL	Access software install script	`GET /phpmyadmin/scripts/setup.php`	12.47
PASSWD	Brute-force password attack	`GET joomla/admin/?uppass=superman1`	2.68
PROXY	Check for open proxy	`GET http://www.wantsfly.com/prx2.php`	0.40
RFI	Look for remote file inclusion (RFI) vulnerabilities	`GET /ec.php?l=http://213.41.16.24/t/c.in`	10.94
SQLI	Look for SQL injection vulnerabilities	`GET /index.php?option=c'`	1.40
XMLRPC	Look for the presence of a certain xmlrpc script	`GET /blog/xmlrpc.php`	18.97
XSS	Check for cross-site-scripting (XSS)	`GET /index.html?umf=<script>foo</script>`	0.19
OTHER	Everything else		8.40

Table 3: Table showing all the attack categories, with a brief explanation, an example, and the fraction of traffic corresponding to the attack in our Web honeypot.

geted at any particular Web applications, but rather at any Web site that is backed by a database. Looking at hacker forums, we find the most popular query for finding sites to try SQLI on is: `inurl:index.php`. Since our honeypot pages are not very highly ranked, we would not be returned by the search engines for such generic queries. Therefore, we are less likely to see traffic from such attackers.

4.3 Comparing Honeypots

In this section, we look at how effective different honeypot setups are in terms of attracting attackers. We consider three scenarios:

1. ***Web server:*** Here, we have just a Web server (in our case, Apache) running on a machine which can be publicly accessed on the Internet. The machine has no hostname, so the only way to access the machine is by its IP address. There are no hyperlinks pointing to the server, so it is not in the index of any search engine or crawler.
2. ***Vulnerable software:*** Here, we install four commonly targeted Web applications, as described in Section 3.2 (a). The application pages are accessible on the Internet, and there are links to them on public Web sites. Therefore, they are crawled and indexed by the search engines.
3. ***Honeypot pages:*** These pages are generated by option (b) as described in Section 3.2. They are simple HTML pages, wrapped in a small PHP script which performs logging. Similar to software pages, the honeypot pages are also crawled and indexed by search engines.

4.3.1 Coverage

All three setups were active for the same duration. Figure 8 shows the effectiveness of each setup. In spite of its relative obscurity, the plain Web server still sees quite a bit of attack traffic. It sees over 1,300 requests from 152 different IP addresses over the course of the three months we had the system running. In fact, we observed the first probe less than four hours after setting up the Web server! This suggests that there are attackers who are continuously scanning IP ranges looking for Web servers. Most of the probes hitting the server were either checking if it allowed proxying, or looking for certain files on the server (which would indicate whether a particular software is installed or running on the machine). These scans can be thwarted by using virtual hosts, and requiring that the correct hostname be provided before the Web server returns anything. However, the availability of Web sites that provide the set of domain names hosted on a particular IP address allows the attacker to easily overcome this hurdle.

The number of IP addresses observed by the vulnerable software and the honeypot pages are roughly proportional, considering we have four software pages and 96 honeypot pages. This shows that our honeypot pages are as good at attracting attackers as the real software. The main trade-off we notice here is that with real software running on the machine, we are able to have many more interactions with a single attacker. On the other hand, with honeypot pages, we are able to reach many more attackers, but the number of interactions with a single attacker is far fewer.

4.3.2 Attacks seen

Table 4 shows which attacks are observed by each setup. We see in general that the heat-seeking honeypot observes instances of almost all the different attacks. The only category which our honeypot does not see is the one where attackers post spam and comments. This seems to be targeted at particular software which support these operations. The software honeypot mainly sees two types of attacks: attackers trying to register accounts and post in the forums and comments (COMMENT), and attackers trying to access the administrator console on these Web applications (ADMIN). The reason the honeypot does not see the same volume of traffic in these categories is because we return responses that the attacker doesn't expect. For example, when the attacker tries to access the admin console, we return a HTTP404 error because the page doesn't exist on our honeypot. How-

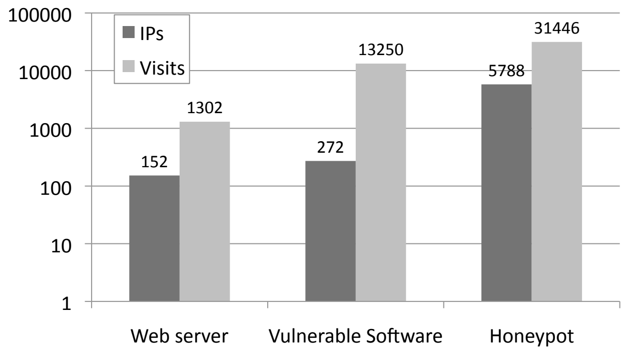

Figure 8: The figure compares the number of total visits and the number of distinct IP addresses, across three setups: Web server, vulnerable software, and the honeypot.

	Traffic in each attack category (%)		
	Apache	Software	Honeypot
ADMIN		47.51	1.00
COMMENT		**49.71**	
FILE			**43.57**
INSTALL	1.94	0.03	12.47
PASSWD	0.85	0.69	2.68
PROXY	**56.62**		0.40
RFI	18.44	0.01	10.94
SQLI		0.01	1.40
XMLRPC			18.97
XSS			0.19
OTHER	22.15	2.05	8.40

Table 4: Table showing the different types of attacks which are seen by each setup. The bold entries indicate the most frequent attack in each setup.

ever, the software honeypot returns an error of HTTP401 or HTTP403; this prompts the attacker to try again. The Web server mostly sees requests for checking if it is an open proxy. In spite of there being no running software, attackers still try to probe for file disclosure and other vulnerabilities.

4.4 Applying whitelists to the Internet

In addition to the logs we obtain from our honeypot, and our software deployment, we also look at other Web servers on the Internet. We first find servers whose HTTP access logs are indexed by search engines. We take a random set of 100 such servers and obtain their access logs. Studying these logs will give us an idea of what kind of attacks are out there in the "wild", and how prevalent they are.Note that these servers are chosen at random, so vary in almost all respects—the popularity, the kinds of software running on them, the number of visitors, etc.

In this section, we look at the volume of attack traffic received by each site. We define a request to be from an attacker, if it tries to fetch a link which isn't whitelisted (i.e., a link which has not been accessed by a crawler). Since this would lead us to falsely flag any dynamically generated links as attack traffic, we place an additional requirement: the requested link should not be present on the server, i.e., the request results in an HTTP404 error. We are therefore conservative in our labeling, and could miss any attacker traffic that succeeded in finding the file or vulnerability it was scanning for. Our estimation of the amount of attack traffic is hence a lower bound. Figure 9 plots what fraction of each site's non-crawler traffic is from attackers. We consider only those sites which had at least 25 visits in all. For 25% of the sites, almost 90% of the traffic came from attackers; i.e., less than 10% of the traffic was due to legitimate visitors. This is mostly the case for small unpopular Web sites that don't have many regular visitors. In the median case, we see that about a quarter of the traffic is from attackers.

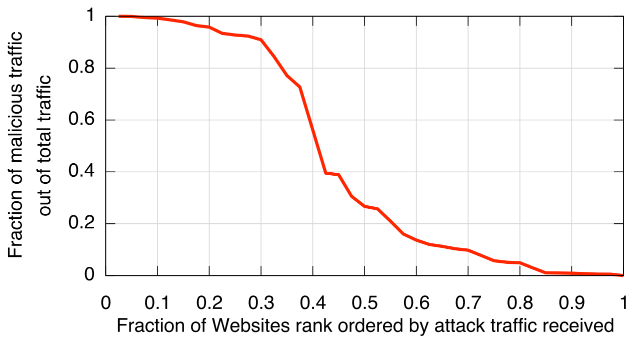

Figure 9: The amount of attack traffic each Web site receives, as a fraction of the total non-crawler traffic received.

5. DISCUSSION

We now discuss certain properties of our system design and ways in which the system could be enhanced to provide greater visibility into attackers' malicious behavior.

Detectability of heat-seeking honeypot

In contrast to client-based honeypots, where malicious binaries could detect the presence of a honeypot environment (such as a virtual machine) and choose to behave differently, attackers cannot easily detect the existence of our honeypot without first talking to our honeypot, which reveals their presence already.

Attracting more attacks

Currently, as our honeypots are set up under a personal homepage, they are not highly ranked in search results and hence miss out on many attack visits. We could get more links from high ranked pages to boost the pagerank in order to gather more attack traffic. Our system could be easily extended to multiple servers in different domains and countries using networks like PlanetLab [5]. These servers would attract attackers with special search restrictions such as domain and language.

Improving reaction times

The main limitation of Web honeypots is the inability to respond quickly to new attacks. When a new application is targeted, the honeypot can put up a related honeypot page immediately after getting the malicious query feed. However, this page will not get attack traffic until some search engines crawl and index it. This delay could be mitigated if the honeypot pages are linked to from popular pages that are crawled frequently and ranked high. To completely solve the problem, though, we need the cooperation of search engines to dynamically include honeypot pages in search results when they detect suspicious queries in realtime. This allows the honeypot to see traffic from attackers early, enabling the detection of emerging threats and 0-day attacks.

Blacklist vs. Whitelist
In this paper, we use a whitelist-based approach to detect malicious traffic. It is different from traditional blacklist-based approaches such as Snort rules [6] and IP blacklists, which usually suffer from false positives, and are sometimes easy to get around if attackers use obfuscation. We see from our honeypots that many of the Web attacks involve accessing a resource that wasn't meant to be accessed. We therefore believe that an approach that uses whitelisting to restrict such probes might help prevent compromise of these Web applications. The advantage of using a whitelist is that by having the Web server implement it, all the different applications running on the server are protected. The main challenge is generation of accurate whitelists, and this might be simplified by using systems like AutoRE to automatically generate the whitelist from a set of allowed requests. Going forward, we believe that whitelists will be an important tool that administrators can use to secure Web applications.

6. CONCLUSIONS

In this paper, we present heat-seeking honeypots, which deploy honeypot pages corresponding to vulnerable software in order to attract attackers. They generate honeypot pages automatically without understanding the latest exploits or knowing which software is affected. During three months of operation, our prototype system attracted an order of magnitude more malicious traffic than vanilla Web servers, and captured a variety of attacks including password guesses, software installation attempts, SQL-injection attacks, remote file inclusion attacks, and cross-site scripting (XSS) attacks. Further, our system can detect malicious IP addresses solely through their Web access patterns, with a false-negative rate of at most 1%. Heat-seeking honeypots are low-cost, scalable tools; we believe that they help understand attackers' targets and methods, and that their use can effectively inform appropriate monitoring and defenses.

7. REFERENCES

[1] Bing. `http://www.bing.com`.

[2] DShield Web Honeypot Project. `http://sites.google.com/site/webhoneypotsite/`.

[3] Glasstopf Honeypot Project. `http://glastopf.org/`.

[4] Most web attacks come via compromised legitimate websites. `http://www.computerweekly.com/Articles/2010/06/18/241655/Most-web-attac%ks-come-via-compromised-legitimate-wesites.htm`.

[5] PlanetLab. `http://www.planet-lab.org/`.

[6] Snort : a free light-weight network intrustion detection system for UNIX and Windows. `http://www.snort.org/`.

[7] Spam SEO trends & statistics. `http://research.zscaler.com/2010/07/spam-seo-trends-statistics-part-ii.%html`.

[8] Google Hack Honeypot, 2005. `http://ghh.sourceforge.net/`.

[9] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling. The Nepenthes platform: An efficient approach to collect malware. In *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 165–184. Springer, 2006.

[10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

[11] D. Eichmann. The rbse spider - balancing effective search against web load, 1994.

[12] J. P. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy. Searching the Searchers with SearchAudit. In *Usenix Security Symposium*, 2010.

[13] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[14] T. Moore and R. Clayton. Evil searching: Compromise and recompromise of internet hosts for phishing. In *13th International Conference on Financial Cryptography and Data Security*, Barbados, 2009.

[15] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, February 2006.

[16] N. Provos. A virtual honeypot framework. In *In Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004.

[17] N. Provos, J. McClain, and K. Wang. Search worms. In *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, pages 1–8, New York, NY, USA, 2006. ACM.

[18] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield. To catch a predator: a natural language approach for eliciting malicious payloads. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 171–183, Berkeley, CA, USA, 2008. USENIX Association.

[19] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[20] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *14th conference on USENIX Security Symposium*, 2005.

[21] Yi-Min Wang and Doug Beck and Xuxian Jiang and Roussi Roussev and Chad Verbowski and Shuo Chen and Sam King. Automated Web Patrol with Strider HoneyMonkeys. In *NDSS*, 2006.

[22] F. Yu, Y. Xie, and Q. Ke. Sbotminer: Large scale search bot detection. In *International Conference on Web Search and Data Mining(WSDM)*, 2010.