Kizzle: A Signature Compiler for Detecting Exploit Kits

Ben Stock
CISPA, Saarland University

Benjamin Livshits
Microsoft Research

Benjamin Zorn
Microsoft Research

***Abstract*—In recent years, the drive-by malware space has undergone significant consolidation. Today, the most common source of drive-by downloads are so-called *exploit kits* (EKs). This paper presents Kizzle, the first prevention technique specifically designed for finding exploit kits.**

Our analysis shows that while the JavaScript delivered by kits varies greatly, the unpacked code varies much less, due to the kits authors' code reuse between versions. Ironically, this well-regarded software engineering practice allows us to build a scalable and precise detector that is able to quickly respond to superficial but frequent changes in EKs.

Kizzle is able to generate anti-virus signatures for detecting EKs, which compare favorably to manually created ones. Kizzle is highly responsive and can generate new signatures within hours. Our experiments show that Kizzle produces high-accuracy signatures. When evaluated over a four-week period, false-positive rates for Kizzle are under 0.03%, while the false-negative rates are under 5%.

I. Introduction

The landscape of drive-by download malware has changed significantly in recent years. There has been a great deal of consolidation in malware production and a shift from attackers writing custom malware to almost exclusively basing drive-by download attacks on exploit kits (EKs) [19]. This approach gives attackers an advantage by allowing them to share and quickly reuse malware components in line with the best software engineering guidelines. It is the natural evolution of the malware ecosystem [12] to specialize in individual tasks such as CVE discovery, packer creation, and malware delivery. We observe that this consolidation, while benefiting attackers, also allows defenders significant opportunities.

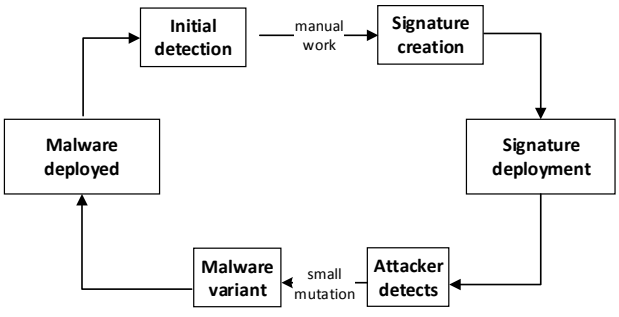


Fig. 1: Adversarial cycle illustrated. There is a built-in asymmetry that Kizzle aims to remedy: the response time for the attacker is only a fraction of that of the defender.

From ad-hoc to structural diversity: The prevalence of EKs changes the diversity in deployed malware in significant ways. While five years ago two deployed variants of the same JavaScript-driven heap spraying exploit might have been written by independent hackers, today much of the time malware variants will come by as a result of malware individualization performed by an EK as well as natural changes in the EK over time.

There is virtually no deployed JavaScript malware that is not obfuscated in some way; at runtime, the obfuscated code is *unpacked*, often multiple times, to get to the ultimate payload. We observe that in practice, much EK-generated malware operates like an onion: the outer layers change fast, often via randomization created by code packers, while the inner layers change more slowly, for example because they contain rarely-changing CVEs.

Arresting the malware cycle: Malware development is chiefly reactive in nature: malware producers create and test their wares against the current generation of readily available detection tools such as anti-virus (AV) engines or EmergingThreats signatures. In a constant arms race with the AV industry, these kits change part of their code to avoid detection by signature-based approaches. While these changes may be distinct between two versions of the same kit, the underlying structure of the fingerprinting and exploitation code rarely changes (as we show later).

This attack-defense pattern is the fundamental nature of the adversarial cycle (Figure 1), as has been noted as early as 20 years ago [23]. Unfortunately, presently, in the drive-by malware space the attacker has a considerable advantage in terms of the amount of work involved. Malware variants are relatively easy to create, most typically by changing the unpacker the attacker uses and testing their variant against AV engines. Indeed, the attacker can easily automate the process of variant creation and testing.

Achilles heel: While the consolidation of malware distribution via exploit kits makes life simpler for the attacker, their strength is also their weakness. Our key detection insight is that code reuse that occurs in EKs over time is their Achilles heel. The core of the unpacked "onion" changes *little* over time and is thus something that can be "tracked" in a sea of unclassified samples with techniques that measure differences between programs.

A. Design Choices – Deployment at Scale

This paper proposes Kizzle, a *signature compiler* that automates the process of synthesizing signatures from captured JavaScript malware variants. Kizzle gives defenders

EK	Flash	Silverlight	Java	Adobe Reader	Internet Explorer	AV check
Sweet Orange	2014-0515		Unknown[1]		2013-2551, 2014-0322	No
Angler	2014-0507, 2014-0515	2013-0074	2013-0422		2013-2551	Yes
RIG	2014-0497	2013-0074	Unknown		2013-2551	Yes
Nuclear exploit kit	(2013-5331), 2014-0497		2013-2423, 2013-2460	2010-0188	2013-2551	Yes

Fig. 2: CVEs used for each malware kit (as of September 2014). The CVEs are broken down into categories.

greater automation in the creation of AV signatures. Our scalable cloud-based implementation allows defenders to generate new signatures for malware variants observed the same day within a matter of hours.

Our design choices are born out of our understanding of the adversarial cycle (Figure 1) in our desire to tip the scale in favor of the defender by reducing the manual effort of the security analyst responsible for writing signatures. Because Kizzle needs to be "seeded" with exploit kits, we are not trying to replace the analyst; we are merely trying to automate parts of the signature generation process to reduce their workload.

Deployment channels: We envision the possibilities of deploying Kizzle in a variety of settings:

- Kizzle signatures may be deployed within a browser, client-side, to scan all or some of the incoming JavaScript code;
- Kizzle signatures can be deployed on the desktop to scan files that are saved to the file system so that if any browser on the machine caches JavaScript files to disk, those files will likely trigger a signature match;
- lastly, Kizzle signatures can be deployed server-side, for instance, a CDN administrator may decide which JavaScript files to host on a CDN in an effort to avoid hosting malware.

It is because of these various deployment scenarios that we choose AV signatures as our *distribution format*. AV signatures enjoy a well-established *deployment channel* with frequent, automatic updates for signature consumers.

At the same time, just about any signature-based scheme and most IDS approaches can be used as an oracle by an attacker trying to evade detection. While we of course recognize this shortcoming, we focus on tipping the balance in favor of the defender in the existing, large-scale signature generation and deployment ecosystem.

Enabling robust deployment at scale: Although we employ machine learning as a core component of our signature-creation strategy in Kizzle, machine learning per se is not our core contribution. We are deliberately using pre-existing off-the-shelf machine learning techniques to reduce the engineering cost and limit the fragility of the end-to-end system. Specifically, we chose to use the DBSCAN clustering strategy.[2]. Kizzle uses existing components to create a multi-stage, distributed clustering system that scales well with the volume of incoming data. In other words, our system can be built and *supported* by security engineers and not machine learning experts.

[2]Clustering has been used in previous malware and intrusion detection research (for example, [28, 40]) but previous work has not identified the opportunity specifically presented by exploit kits for clustering or defined an effective clustering strategy in the presence of packing and obfuscation in the delivered JavaScript.

B. Contributions

This paper makes the following contributions:

Insight: Through detailed examination of existing EKs we document the evolution of these kits over time, noting how they grow and evolve. In particular, EKs often evolve by appending new exploits, the outer packer changes without major changes to the inner layer, and different EK families "borrow" exploits from each other. These observations suggest that there is a great deal of commonality and code reuse, both across EK versions, and between the different EKs, which enables EK-focused detection strategies.

Clustering in the cloud: Based on these observations we built a high-performance processing pipeline for preprocessing a large number of "grayware" JavaScript samples into a structured token stream and parallelizing the process of clustering to be run in the cloud. Lastly, the clusters are matched with known exploit kits for both marking them as either benign or malicious, and kit identification.

Signature generation: Out of the detected code clusters, we propose a simple algorithm for quickly automatically generating *structural signatures* which may be deployed within an anti-virus engine. Our approach, dubbed Kizzle, produce signatures which are comparable in quality and readability to those a human analyst may write. With these structural signatures, whose accuracy rivals those written by analysts, we can track EK changes in *minutes* rather than days. Kizzle signatures can be also deployed within the browser, enabling fast detection at JavaScript execution runtime.

Evaluation: In this paper we primarily focus on detecting four popular exploit kits. Our focus is supported by a recent analysis performed by ZScaler ThreatLab that concluded that three out of four kits we analyze are "the top Exploit Kits that we have seen involved in various Malvertising campaigns in 2015 [42]." Similar findings have also been discussed by TrendMicro [39]. Figure 2 shows a brief summary of information about these EKs. Our month-long experimental evaluation shows that automatically-produced structural signatures are comparable to those produced manually. We compare Kizzle against a widely used commercial AV engine and find that it produces comparable false positive and false negative rates for the

exploit kits we targeted. With respect to our evaluation, false positive rates for Kizzle are under 0.03%, while the false negative rates are under 5%.

C. Paper Organization

The rest of this paper is organized as follows. Section II gives some background on exploit kits and how they are typically constructed. Section III discusses the technical details of Kizzle. Section IV contains a detailed experimental evaluation. Section V talks about the limitations of our approach. Finally, Sections VI and VII summarize related work and conclude.

II. Background

The last years have witnessed a shift from unique drive-by downloads to a consolidation into exploit kits, which incorporate a variety of exploits for known vulnerabilities. This has several advantages for all malicious parties involved. As mentioned in the Microsoft Security Intelligence Report V16, "Commercial exploit kits have existed since at least 2006 in various forms, but early versions required a considerable amount of technical expertise to use, which limited their appeal among prospective attackers. This requirement changed in 2010 with the initial release of the Blackhole exploit kit, which was designed to be usable by novice attackers with limited technical skills."

Exploit kits bring the benefits of specialization to malware production. A botnet herder can now focus on development of his software rather than having to build exploits that target vulnerabilities in a browser and plugins. On the other hand, the maintainer of a single exploit kit may use it to distribute different pieces of malicious software, optimizing his revenue stream. This trend is also shown by glimpses security researchers sometimes got into the backend functionality and operation of such kits, such as detailed information on the rate of successful infection from the kits [15, 37]. Interested readers are referred to [19] for a more comprehensive summary. Note, however, that most up-to-date information can be found via blogs like SpiderLabs[3] and "Malware don't need Coffee[4]", which are updated regularly as exploit kit updates emerge.

Focus of this paper: As can be seen from Figure 2, the EKs under investigation are targeting vulnerabilities in five browser and plugin components.

An interesting observation in this instance is the fact that Nuclear contains an exploit targeting a CVE from 2010 in Adobe Reader, highlighting the fact that exploitable vulnerabilities are hard to come by. Note that in September 2014, three of the exploit kits used the exact same code to check for certain system files belonging to AV solutions. To avoid detection, the exploit kits do not attempt to target a flaw when such files are found.

Our investigation focuses on these four exploit kits because they were the top kits appearing in our JavaScript samples and were in active development during our measurement window. Figure 5 shows how one of these kits, Nuclear, experienced near-constant updates during the time period of our study, making it a good subject for studying malware evolution. While other kits exist, relatively few are prominent and in active development at a given time, and we consider these representative of the broader population.

A. Exploit Kit Structure

Exploit kits are comprised of several components organized into layers (Figure 3), that typically include an unpacker, a plugin and AV detector, an `eval`/execution trigger, and, at least one malicious payload.

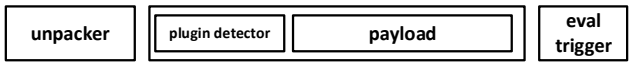


Fig. 3: Structure and components of a typical EK.

An exploit kit usually incorporates a range of CVEs that it attempts to exploit, which target a variety of operating systems, plugins, and browsers. In the following, we discuss these components, showing how they evolve over time.

Unpackers: The outer-most layer of this onion is typically used to ensure that a security analyst or a web site administrator cannot effortlessly determine the inner workings of the exploit kit. This can either be achieved by packing the underlying pieces or by at least applying obfuscation techniques such as string encoding.

Figure 4 shows samples of packers from the RIG and Nuclear explot kits, highlighting the differences between families. While Nuclear relies on an encryption key that is used when unpacking the malicious payload, RIG uses a buffer which is dynamically filled during runtime with the ASCII codes for the payload, intermixed with a delimiter. We found that this delimiter is randomized between different versions of the kit. In contrast, the encryption key — and therefore the encrypted payload — for the Nuclear exploit kit differs in every response, highlighting the fact that it is difficult to pattern-match on obfuscated code.

Malicious payload: The actual payload typically targets a set of known vulnerabilities in the browser or plugins. As can be seen from the Exploit Pack Table[5], 5–7 CVEs per kit is fairly typical. The payload is often interleaved with plugin detection code, e.g., an HTML element is added to the DOM pointing to a malicious flash file if a vulnerable version was detected previously.

Eval trigger: After the malware is fully unfolded, there is usually a short trigger that starts the EK execution

[2]Note that for some of the kits, while a Java exploit was present, no version checking was conducted by the kit, thus determining the specific CVE is difficult if not impossible.

[3]`http://blog.spiderlabs.com/`

[4]`http://malware.dontneedcoffee.com/`

[5]`http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html`

```
1  var buffer="";
2  var delim="y6";
3
4  function collect(text) {
5   buffer += text;
6  }
7
8  collect("47y642y6100y6");
9  collect("102y6103y6104..");
10
11 pieces = buffer.split(delim);
12
13 screlem = document.createElement("script");
14
15 for (var i=0; i<pieces.length; i++) {
16  screlem.text += String.fromCharCode(pieces[i]);
17 }
18
19 document.body.appendChild(screlem);
```

(a) RIG

```
1  var payload =
2      "691722434526012276437
3      188215239887038218 8197
4      642634057014376927 6221
5      2757616434526211272.."
6  var cryptkey =
7      "Io^Rg_U8$\\ep6kAu.rVvn!'Ti15SQqd-
8      #2@\"{(l4xcbt?>[3E/sP:O<D7*yz|m+Z;JBf)
9      hX9Gw LOCF%KN},&YaMHj=]W";
10 ...
11
12 getter = function(a){
13   return a;
14 };
15
16 thiscopy = this;
17 doc = thiscopy[thiscopy["getter"]("document")]
18 bgc = doc[thiscopy["getter"]("bgColor")];
19
20 evl = thiscopy["getter"]("ev#333366al")
21 win = thiscopy["getter"]("win#333366dow")
22
23 thiscopy
24     [win["replace"](bgc,"")]
25     [evl["replace"](bgc, "")](payload);
```

(b) Nuclear exploit kit

Fig. 4: Two typical code unpackers from exploit kits.

process. Examples of these triggers are shown on line 19 of Figure 4(a) and lines 23–25 of Figure 4(b).

B. Evolution of an Exploit Kit

To understand how EK components evolve in the wild, we captured samples of Nuclear over the course of three months and tracked changes to the kit. We summarize some of the mutation approaches below.

Changing the packer: Figure 5 illustrates specific changes that were made to packer and payload in the kit to avoid AV detection on a time line. Many of the changes were very local, changing the way that the kit obscured calls to `eval`. For example, between 6/1 and 6/14, the attacker changed `ev#FFFFFFal` to `e#FFFFFFFval`. Over the course of the three months, we see a total of 13 small syntactic changes in this category. Only one of these packer changes (on 8/12) changed the semantics of the packer.

Appending new exploits: We observed changes to the other components of the kit, i.e., plug-in detection and payload, occur much less frequently. On 7/29, AV detection was added to the plug-in detector, and on 8/27 a new CVE was added. Nothing was removed from either the plug-in detector or the payload over this period. This supports our claim that EKs change their outer layer frequently to avoid AV detection, but typically modify their inner components by appending to them and even then only infrequently.

Code borrowing: A noteworthy fact in this instance is that in June, the Nuclear exploit kit did not utilize any code aiming at detecting AV software running on the victim's machine. We initially observed this behavior in the RIG exploit kit starting in May. The exact code we had observed for RIG was used in Nuclear from August, apparently having been copied from the rivaling kit.

While we use Nuclear exploit kit as a specific example here, we observed similar changes in all the exploit kits we studied. In summary, we observe that kits typically change in three ways, namely **changing** the unpacker (frequent), **appending** new exploits (infrequent), and **borrowing** code from other kits (infrequent).

C. Adversarial Cycle

Exploit kit authors are in a constant arms race with anti-virus companies as well as rivaling kits' authors. While on the one hand, the kits try to avoid detection by anti-virus engines, their revenue stream is dependent on the amount of machines they can infect. Therefore, kit authors always try to include multiple exploits, and if one kit includes a new exploit, we observed that these exploits are quickly incorporated into other kits as well. As we have seen in the example of the Nuclear exploit kit above, kit authors attempt to avoid detection by anti-virus engines by modifying the code, whereas in turn AV analysts try to create new signatures matching the modified variants. This process can be abstracted to the adversarial cycle shown in Figure 1.

Initially, an exploit kit is not detected by AV, which presents a very challenging problem for an analyst. First they have to find examples of the undetected variant and then need to create a new signature which matches this undetected variant, trading off precision and recall – i.e., the signature has to be able to catch all occurrences of the exploit kit while not blocking benign components. Naturally this takes time and effort and despite this cost, AV engines update their signatures frequently to keep pace with the malware writers. After an analyst is satisfied they have a precise and unique signature, they deploy it.

At this point, the attacker responds, determining that his kit is now detected by deployed AV signatures. For the attacker, this can be learned automatically, e.g., by submitting a URL containing his kit to an AV scanner. Once this step has occurred (left-hand side of the figure), he takes measures to counter detection, such as slight modification to the part of the code that would be suspicious (e.g., calls to `eval`), or, in more drastic cases we observed in the wild, exchanging entire pieces, such as the unpacker.

Depending on the type of change, this task can be easily accomplished within minutes — and more importantly, an attacker can scan his code with an AV solution to determine if it now passes detection, giving him an advantage over vendors, who need to find ways to detect the new

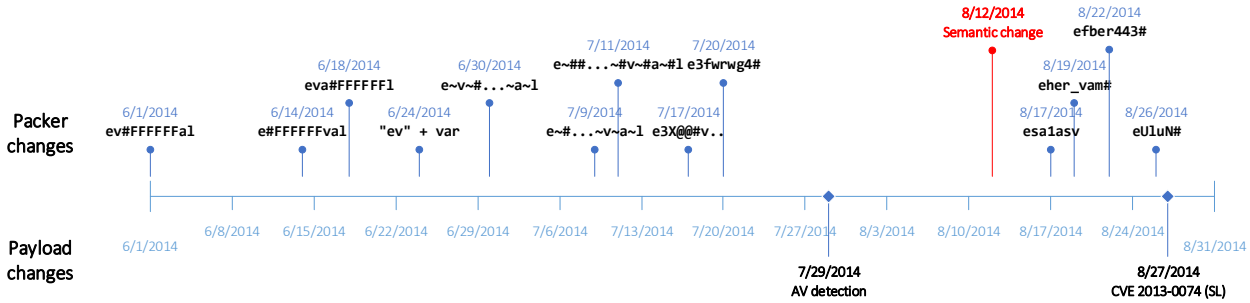


Fig. 5: Evolution of the NUCLEAR EXPLOIT KIT over a three-month period in 2014. In this timeline, packer changes are shown above the axis and payload changes below the axis. The lion's share of changes are superficial changes to the packer.

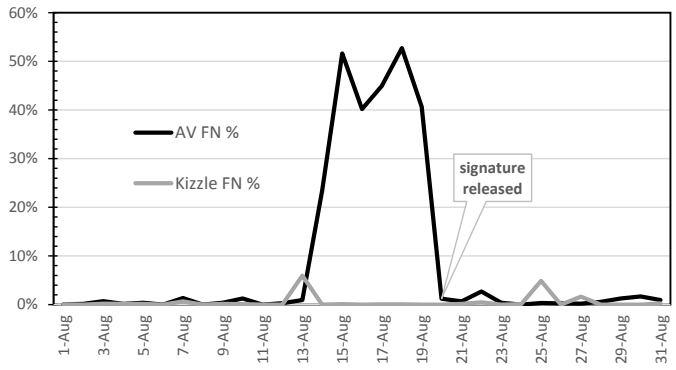


Fig. 6: Window of vulnerability for ANGLER in August, 2014 for a commercial AV engine. The window starts around August 13th and continues to roughly August 19th.

variant of the kit. This shows the imbalance between the involved parties, i.e., the effort and reaction time of the kit author is much lower than that of the AV vendors.

Example 1 Angler in August Figure 6 shows false negatives for the ANGLER exploit kit in the month of August 2014 for a widely used commercial AV engine[6]. By observing the changes to the kit over this time, we understand what happened. Before August, 13th, the exploit kit contained an HTML snippet that included a Java exploit with a specific unique string on which the AV signature matched. On August, 13th, the string on which the signature matched was incorporated into the obfuscated body of kit and only written to the document if a vulnerable version of Java was installed on the system. This change resulted in a window of time when variants of the kits were undetected. □

III. TECHNIQUES

In this section, we describe the implementation of KIZZLE, which is illustrated at a high level in Figure 7. For a more detailed discussion on the algorithms, we refer the reader to our accompanying technical report [38]. The input to KIZZLE is a set of new samples and a set of existing unpacked malware samples which correspond to exploit kits KIZZLE is aiming to detect. The algorithmic elements of KIZZLE include abstracting the samples into token sequences, clustering the samples, labeling the clusters, and generating signatures for malicious clusters.

Main driver: The processing starts with a new collection of samples, whereas a sample consists of a complete HTML document, including all inline script elements. The main routine breaks the new samples into a set of clusters, labels each cluster either as benign or corresponding to a known kit, and if the cluster is malicious, generates a new signature for that cluster based on the samples in it. We consider each of the parts of the task in turn in the subsections below.

A. Clustering Samples

The process of clustering the input samples is computationally expensive, and as a result, benefits from parallelization across a set of machines. The first stage in our process is to randomly *partition* the samples across a cluster of machines.

For each partition, the samples are tokenized from the concrete JavaScript source code represented as Unicode to a sequence of abstract JavaScript tokens that include `Keyword`, `Identifier`, `Punctuation`, and `String`. Figure 8 gives an example of tokenization in action.

We cluster the samples based on these token strings in order to eliminate artificial noise created by an attacker in the form of randomized variable names, etc. We apply a hierarchical clustering algorithm, specifically DBSCAN [11], using the edit distance between token strings as a means of determining the distance between any two samples.

We experimentally determined that a threshold of 0.10 is sufficient to generate a reasonably small number of clusters, while not generating clusters that are too generic, i.e., contain samples that do not belong to the same family of malware (or snippets of benign code). In the reduction phase, the clusters determined by each partition are combined in a final step.

Next, we consider each distinct cluster, selecting a single prototype sample from the cluster, unpacking it (if it is packed) and then attempting to label it. This unpacking step can be conducted by hooking into the `eval` loop of the JavaScript engine [7]. For our work, which focuses on

[6]We anonymize the exact engine for two reasons: first, we believe that all engines exhibit the same behavior despite constant efforts by the analysts to keep them current, and second because EULA agreements typically prevent disclosing such comparisons.

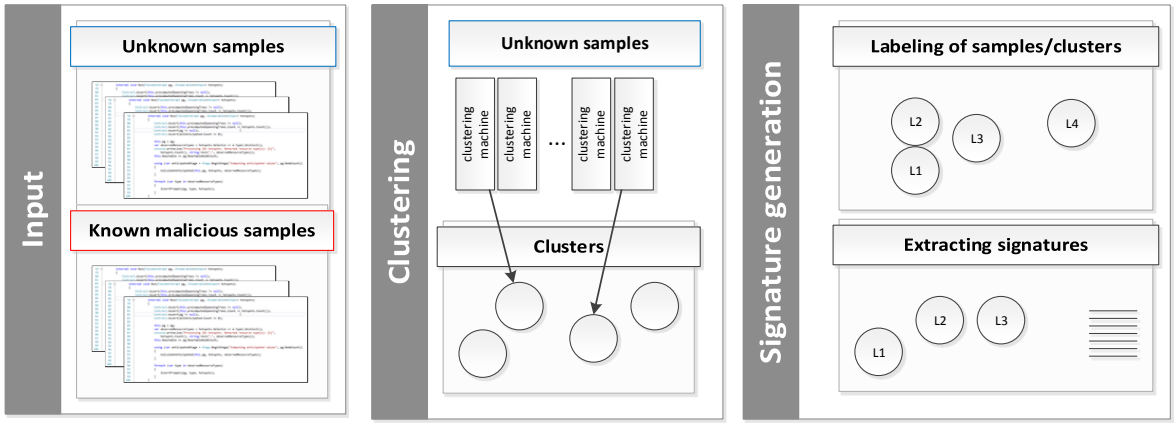


Fig. 7: Architecture of KIZZLE.

Token	Class
`var`	Keyword
`Euur1V`	Identifier
`=`	Punctuation
`this`	Identifier
`[`	Punctuation
`"l9D"`	String
`]`	Punctuation
`(`	Punctuation
`"ev#333399al"`	String
`)`	Punctuation

Fig. 8: Tokenization in action.

a fixed set of exploit kits, we instead implemented unpackers for all kits under investigation. With the resulting, unpacked cluster prototype, we label the cluster.

B. Labeling Clusters

To label clusters with their corresponding EK family label, we use winnowing [34], a technique originally proposed for detecting plagiarism in code. Using a collection of known unpacked malware samples (with exploit family labels), we generate a winnow histogram for the cluster prototype and compare it against the winnow histograms for all the known malware samples. If there is sufficient overlap (based on a threshold that we determined empirically is malware family specific), we then consider the cluster represented by the prototype to be malicious and from the corresponding family.

C. Signature Creation

For each cluster that is labeled as malicious, we generate a signature from the packed samples in that cluster with the following method.

The first step in signature creation is to find a maximum value of N such that every sample in a cluster has a common token string subsequence of length up to N tokens. We cap this maximum length at 200 tokens. We find this subsequence with binary search, varying N, and determining if a common subsequence of length N exists. An additional constraint, imposed during the search for a common subsequence, is that it is unique in every sample.

Once the length of the common subsequence is known and sufficiently long (short sequences are discarded), the exact sequences of tokens and characters in each of the samples from the malicious cluster are extracted. In addition, for each offset in the token sequence, the algorithm determines the distinct set of concrete strings found in the different samples at that token offset.

Figure 9 illustrates this step, showing a cluster with three samples and the process of determining the distinct values at each offset. Note, that although the original string contains quotation marks, these are automatically removed by AV scanners in a normalization step. Therefore, we omit them in the final signature. Finally, after gathering all variants of a token at each offset, the algorithm determines a regular expression-based signature, one token at at time.

If the value is the same across *all* samples, our algorithm adds the concrete value to the signature. Otherwise, the algorithm must generate a regular expression that will match all elements of this set. While this is a well-studied problem in general (The $L^\star$ algorithm [2] can infer a minimally accepting DFA), we implement an approach focusing on our expectations of the kinds of diversity malware writers are likely to put into their code.

We compute an expression that will accept strings of the observed lengths, and containing the characters observed by drawing on a predefined set of common patterns such as `[a-z]+`, `[a-zA-Z0-9]+`, etc. The current approach uses brute force to determine a working pattern, but a more selective approach could build a more efficient decision procedure from the predefined templates.

Example 2 KIZZLE signatures. Figure 10 shows generated signature for the NUCLEAR and SWEET ORANGE kits. For the first signatures, KIZZLE picked up on the strings delimited by `Ulun`. While such long strings, that do not naturally occur in benign applications, make the creation of a signature easy, the kit author can easily change these

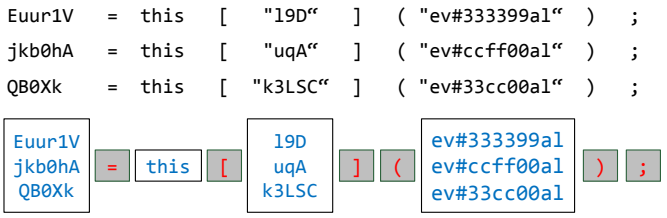


Fig. 9: An example of signature generation in action.

```
(?<var0>[0-9a-zA-Z]{3,6})=\[\[(?<var1>
[0-9a-zA-Z]{3,6})\[(?<var2>[0-9a-zA-Z"']
{5,8})]\("cUluNoUluNnUluNcUluNaUluNtUluN"\),
\k<var1>\[\k<var2>]\("sUluNuUluNbUluNsUluNtUluNrUluN"\),
\k<var1>\[\k<var2>]\("dUluNoUluNcUluNuUluNmUluNeUluNnUluNtUluN"\),
\k<var1>\[\k<var2>]\("CUluNoUluNlUluNoUluNrUluN"\),
\k<var1>\[\k<var2>]\("lUluNeUluNnUluNgUluNtUluNhUluN"\)],
\[\k<var1>\[\k<var2>]\((?<var3>.{57})\),\k<var1>
\[\k<var2>]\((?<var4>.{67})\),\k<var1>
\[\k<var2>]\("rUluNeUluNpUluNlUluNaUluNcUluNeUluN"\)]]
var(?<var5>[0-9a-zA-Z]{3,7})
```

(a) Nuclear exploit kit

```
\)\)\)\{varaa=xx\.join\(""\)ar\[\(Math\.exp\(1\)-Math\.E\)]
\[\(1\)\*2]="l"ar\[\(1\)]\[3]="WWWWWWWbEWsjdhfW"varq=
\(Math\.exp\(1\)-Math\.E\)for
\(qq<ar\[\(Math\.exp\(1\)-Math\.E\)]\.length\+
\+q\)\{aa=aa\.cnvbsdfYTQUWETQWUEASA\(newRegExp\(ar\[\(1\)]
\[q\+\(1\)],"g"\),ar\[\(Math\.exp\(1\)-Math\.E\)]\[q]\)\}
returnaa\}return""\}function(?<var0>[a-zA-Z]{6})\(\)
\{varok=\[(?<var1>[0-9a-zA-Z"']{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var2>[0-9a-zA-Z"']{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var3>[0-9a-zA-Z"']{17})\.charAt\(Math\.sqrt
\(196\)\),(?<var4>[0-9a-zA-Z"']{21})\.charAt\(Math\.sqrt
\(324\)\),(?<var5>[0-9a-zA-Z"']{21})\.
```

(b) Sweet orange

Fig. 10: Examples of KIZZLE-generated signatures.

to circumvent a matching signature. Since, however, KIZZLE generates these automatically, this advantage vanishes.

Also, KIZZLE picked up on the usage of templatized variable names, as can be observed by the combination of `var1` and `var2` in lines 4 to 9 of NUCLEAR. While SWEET ORANGE does not use such delimiters, it uses a simple obfuscation technique, namely exchanging static integer values with calls to the `Math.sqrt` function, allowing it to simply change this obfuscation by using other mathematical operations. Again, KIZZLE picked up on this property of the packed payload, generating a precise signature. □

IV. EVALUATION

To evaluate KIZZLE, we gathered potentially malicious samples using a browser instrumented to report telemetry collected by Internet Explorer from pages that have ActiveX content. The pages sampled came from a broad spectrum of URLs representing pages that typical users might visit. We worked with an anti-malware vendor to hook into the `IExtensionValidation` interface (`http://msdn.microsoft.com/en-us/library/dn301826(v=vs.85).aspx`) for data extraction. The `Validate` method of this interface allows the capture the underlying HTML and JavaScript. Because we sample data during a potentially suspicious operation (loading ActiveX content), the fraction of malware we see is likely to be substantially higher than a typical browser will see, hence we consider our data stream "grayware". In total, we gathered data for a month (August 2014) and captured between 80,000 and 500,000 samples per day, i.e., several gigabytes of JavaScript code for KIZZLE to process daily. Another viable source of likely malicious samples would be a malware analysis system such as Wepawet [6] or VirusTotal.com.

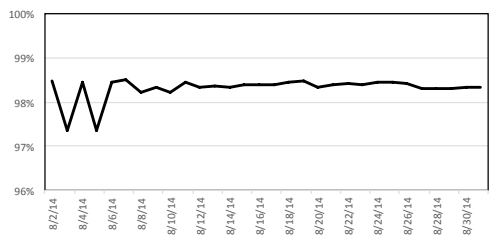
Throughout the rest of this section, we focus on the four exploit kits that are most prevalent in our data: NUCLEAR, SWEET ORANGE, ANGLER, and RIG. These are the same kits highlighted in recent evaluations performed by ZSCaler ThreatLab [42] and TrendMicro [39]. All these kits follow the pattern we describe in Section II: they are packed on the outside and are relatively similar when unpacked. We compare KIZZLE with a state-of-the-art commercial AV implementation, which we anonymize to avoid drawing generalizations based on our limited observations (our position is that all commercial AV vendors have similar challenges).

Experimental Setup: Figure 11 shows our measurements of how these kits change over the course of a month. We measure the overlap between the unpacked centroids of malicious clusters on each day with centroids of the clusters of all previous days based on winnowing (Section III) and report the maximum overlap. Figure 11 shows that, for three of the four kits, the amount of change over the course of the entire month is quite small, often only a few percent. This contrasts greatly from the *external* changes at the level of the packed kits as shown in Figure 5, which happen every few days.

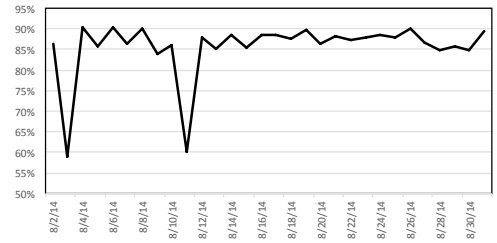
These observations confirm our hypothesis that most of the change is *external* and happens on the packer that surrounds the logic of the kit. In the case of NUCLEAR EXPLOIT KIT, there is very little change at all. We do note that RIG (Figure 11(d)) is an outlier, showing changes of 50% from day over day. This behavior is explained by noticing that the changes reflect modifications of the embedded URLs of the kit, and, given the body of the kit is relatively short, these URLs alone represent a significant enough part of the code to create a 50% churn.

Cluster-Based Processing Performance: Our implementation is based on using a cluster of machines and exploiting the inherent parallelism of our approach. For the performance numbers found in this section, we used 50 machines for sample clustering and one machine for the final signature generation.

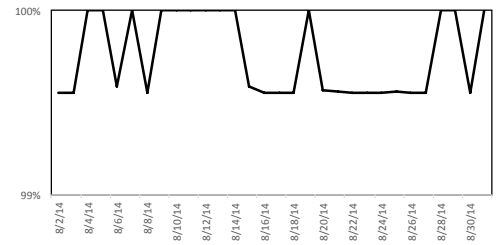
Our experience shows that clustering takes the majority of this time, as opposed to signature generation. The reduce step described in Section III-A is often the bottleneck when we needed to reconcile the clusters computed across the distributed machines. With more effort, we believe that the reduction step can be parallelized as well in future work to improve our scalability. In practice, our runs consistently completed in about 90 minutes when processing



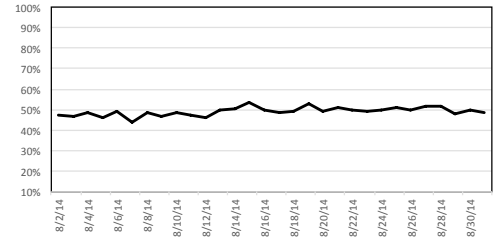
(a) Nuclear



(b) Sweet Orange



(c) Angler



(d) RIG

Fig. 11: Similarity over time for a month-long time window. Note that the y axis is *different* across the graphs: sometimes, the range of similarities is very narrow.

the daily data. We find this processing time is adequate and can be further improved with more machines.

In our experiments, we detected between 280 and 1,200 clusters per day. However, almost all of them correspond to benign code, while only a handful are detected as malicious and labeled as one of the exploit kits in question. This shows that despite the fact that we are analyzing grayware, much of what we observe is benign code that falls into a relatively small number of frequently observed clusters.

Figure 10 shows some examples of the signatures produced by Kizzle. Overall, two things immediately stand out for the automatically-generated signatures: they are long and they are very specific. Both of these characteristics contribute to Kizzle signatures being less prone to false positives, as we show in Figure 13.

Signature effectiveness: Figure 12 shows the lengths of Kizzle-generated signatures over the month-long period of time. We show the length of the signatures, in characters, on the y axis of the graph. Every time there is a "bump" in the line for one of the EKs, this means that Kizzle decides to create a new signature.

To help the reader correlate Kizzle signatures with manually-generated signatures, we show the labels for hand-crafted signatures created to address these EKs over the same period of time. To highlight some of the insights, consider Nuclear exploit kit (blue line) starting on August, 17th. As a packing strategy, the Nuclear exploit kit uses a delimiter, which separates characters `ev` and `al` and `win` and `dow` in Figure 4(b). You can also spot this delimeter-based approach in Figure 10 where a string like `sUluNuUluNbUluNsUluNtUluNrUluN` unpacks into `substr`. The strategy that Nuclear exploit kit uses is to change this delimiter frequently because the malware writer suspects that AV signatures will try to match this code. The green oval call-outs in the figure show signature-avoiding changes to the delimiter as part of kit evolution.

The Kizzle algorithm can immediately react to these minor changes in the body of the kit, as indicated by daily changes in Kizzle signatures. To contrast with manually-produced signatures, the first AV signature that we see responding to these changes emerges on August 25^{th}. Note that it may be the case that the AV signature did not need to be updated to be effective, but the figure illustrates that Kizzle will automatically respond to kit changes daily. Figure 10 shows an example of two signatures produced by Kizzle.

Precision of Kizzle-Generated Signatures: The quality of any anti-virus solution is based on its ability to find most viruses with a very low false positive rate. Our goal for Kizzle is to provide rates comparable to human-written AV signatures when tested over the month of data we collected. Figure 13 shows false positive and false negative rates for Kizzle compared to those for AV. Overall, false positive rates are lower for Kizzle (except for a period between August 24 and August 26th). Figure 15 shows a representative false positive. False positive rates for Kizzle overall are very small, i.e., under 0.03%.

Figure 14 shows some details of our evaluation. The kit that gave Kizzle the most challenge was RIG, which occurred with low frequency in our sample set. RIG also changed more on a daily basis, as illustrated in Figure 11.

Ground Truth: To approximate the ground truth, we took the *union* of samples matched by both AV signatures and the Kizzle approach and examined the overlap. Subsequently, to confirm false positives and false negatives, we manually inspected approximately 7,000 files, using some scripting automation to bucket samples together. In addition, we analyzed the compliment of that union for

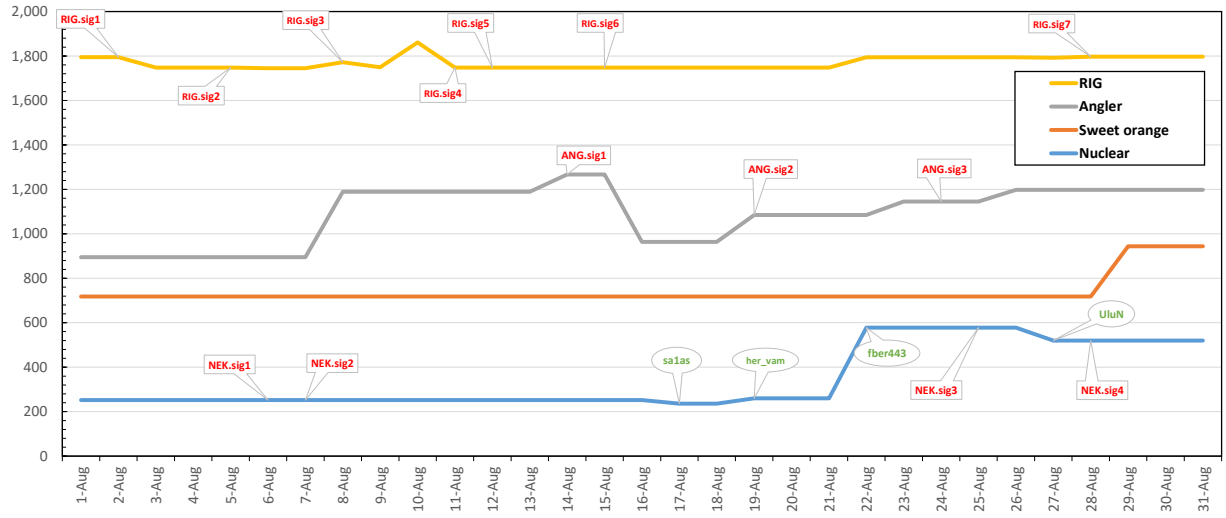
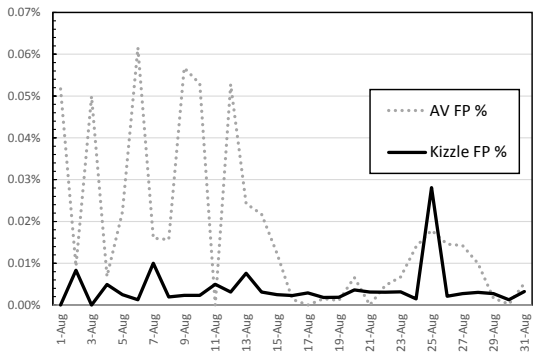
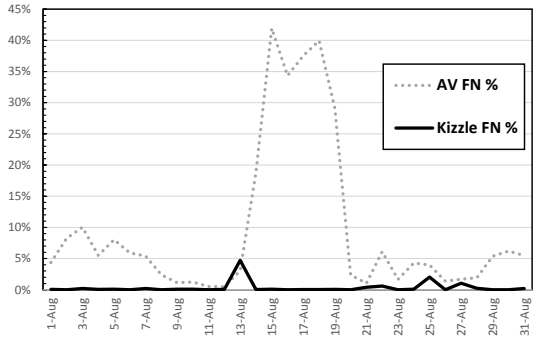


Fig. 12: Signature lengths over time for a month-long time window. Red call-outs are new signatures issued by AV. Green oval call-outs show delimiter changes in NUCLEAR EXPLOIT KIT.



(a) False positives over time for a month-long time window.



(b) False negatives over time for a month-long time window.

Fig. 13: False positives and false negatives over time for a month-long time window: KIZZLE vs. AV.

URL patterns matching known exploit kits. In doing so, we found no cases in which a URL indicated that there were malicious samples in this compliment. In total, this tedious manual validation took approximately 15 hours.

Figure 13 shows that false negative rates for KIZZLE are smaller than those for AV as well. This shows that KIZZLE successfully *balances* false positive and false negative requirements. In particular, there is a spike in false negatives

EK	Ground truth	AV FP	AV FN	KIZZLE FP	KIZZLE FN
NUCLEAR	6,106	1	1,671	25	8
SWEET ORANGE	11,315	0	2	0	1
ANGLER	40,026	635	4,213	0	196
RIG	1,409	11	30	241	144
Sum	58,856	647	7,587	266	349

Fig. 14: False positives and false negatives: absolute counts comparing KIZZLE vs. AV.

between August, 13th, and August, 21st, for AV which KIZZLE does not suffer from. The majority of these false negatives are due to AV's failure to detect a variant of ANGLER (Figure 6).

Overall, the false negative rate for KIZZLE is under 5% for the month of August. A weakness of our approach is that, if a kit changes *drastically* overnight, KIZZLE may no longer be able to connect previous samples to subsequent ones. In practice this did not occur during the period of time we examined because kit authors reuse either the unpacked body of the kit (Figure 11), or because they reuse the packer. When we experience false negatives, it is generally because of changes in the kit that are not numerous enough in our grayware stream to warrant a separate cluster. An example of this is a small bump in false negatives for ANGLER on August, 13th, in Figure 6, which produced some, but not enough new variants of ANGLER for KIZZLE to produce a new signature.

V. DISCUSSION

Our approach combines automatically building signatures on the packed versions of exploit kits by reasoning about maliciousness based on comparing the unpacked versions to previous known attacks.

Choice of EKs: Our results are based on studying the behavior of four exploit kits over a period of one month.

```
\},toString:\(\{\}\)\.constructor\.prototype\
.toString,isPlainObject:function\(c\)\{vara=
this,bif\(!c\|\|a\.rgx\.any\.test\(a\.toString\
.call\(c\)\)\|\|c\.window==c\|\|a\.rgx\.num\
.test\(a\.toString\.call\(c\.nodeType\)\)\)\
{return0\}try\{if\(!a\.hasOwn\(c,"constructor"\)
&&!a\.hasOwn\(c\.constructor\.prototype,
"isPrototypeOf"\)\)\{return0\}\}catch\(b\)\
{return0\}return1\},isDefined:function\(b\)\
{returntypeofb!="undefined"\},isArray:
function\(b\)\{returnthis\.rgx\.arr\.test\
(this\.toString\.call\(b\)\)\},isString:
function\(b\)\{returnthis\.rgx\.str\.test\
(this\.toString\.call\(b\)\)\},isNum:function\(b\)
```

Fig. 15: A false positive for KIZZLE extracted from `PluginDetect`; it shares a very high (79%) overlap with NUCLEAR EXPLOIT KIT.

We have looked at the same kits over a longer period (and observed consistent behavior) and these kits do represent a significant fraction of all malware we observe in our data stream, but our experiments are still limited. Our choice of kits mostly coincides with the most important EKs of 2014 and 2015, as identified by a different security companies [39, 42], and we believe our results will carry over to other EKs, given that they employ a similar packing strategy.

Tuning the ML: As with any solution based on clustering and abstracting streams of data, a number of tuning knobs control the effectiveness of the approach. For example, how many samples do we need to define a cluster, how long should the generated signatures be, etc. Finding the right values for these parameters and adjusting them due to the dynamic nature of a malicious opponent makes keeping such a system well-tuned challenging. Likely, observing detection accuracy over time as part of operations with the attacker adjusting to KIZZLE is needed.

EK structure: Our approach is based on the fact that while the effort to change the outer layer of an EK is relatively small, changing the syntax, yet not the semantics, of the inner layer is non-trivial. However, research by Payer has highlighted that for binary files, automated rewriting to achieve just such syntactic changes is feasible [27]. While that work aims specifically at binaries, adoption of such automated rewriting schemes would impair KIZZLE's ability to track the kits in their unfolded form, if exploit kits move closer to a "fully polymorphic" model.

Deployment and avoidance: As shown in Figure 1, there is an arms race between created signatures and avoidance by an adversary. KIZZLE's signatures, not unlike hand-crafted AV signatures, can be circumvented by simple trial-and-error—the attacker loads a URL contain his packed kit and checks whether the AV flags it. If the AV triggers, he can adjust the code to see if that change allows him to bypass detection. This allows him to produce a new, undetected variant of his kit. However, the inner-most layer is not as easy to change, as the code is often not even originating from the kit author, but taken from another kit. Therefore, even though the new variant has no resemblance to the previous versions on the outside, they will most likely overlap in the inner-most code, allowing KIZZLE to correctly label the resulting clusters as malicious and, thus, producing a signature to match the new variant. This process is fully automated and therefore does not require any manual work on the side of the analyst. In contrast, the malware author is now faced with a signature capable of detecting his changed EK variant, requiring him to change the packer again.

One means for an attacker to bypass detection is to change the inner-most part of the code to such an extend that KIZZLE is no longer able to correctly classify a malicious cluster. This step, however, cannot be automated as easily since KIZZLE does not provide an attacker with the means of directly checking a given sample for detection. Rather, he has to create a new variant with major changes to the inner core and wait for KIZZLE to create (or not create) a signature. To counter such attacks, KIZZLE can be extended to employ *hidden signatures* on the server side. Such signatures can either match on specific strings contained in the inner layer or even match on execution behavior. As they never leave the server, the adversary has no means of learning what they match on and, thus, is not able to circumvent detection. The current generation of EKs, however, does not change the unpacked code in such a manner and, thus, we opted not to implement such a detection mechanism.

An attacker aware of the signature creation algorithm can try to modify his packer such that our algorithm fails. An example for this is the insertion of a random number of superfluous JavaScript instructions between relevant operations to beat the structural signatures. We believe, however, that our approach can be extended to create signatures which not only match *one* consecutive token sequence, but rather consist of multiple, shorter sequences.

VI. RELATED WORK

We cover three most closely related areas of research.

Exploit Kits: Previous work on exploit kits has focused mainly on examining their server-side components. Kotov *et al.* analyze the server-side code for 24 (partially inactive) different families and found that 82% of the analyzed kits use some form of obfuscation [19]. Additional research into the server-side components has been conducted by De Maio *et al.* [8]. The authors conclude that several of the analyzed kits are based on one another. They are able to produce combinations of both `User-Agent`s and GET parameters such that an infection is more likely to occur.

Grier *et al.* [12] have conducted an analysis into the effects exploit kits have on the malware ecosystem, finding that the analyzed kits are used to deliver several different families of malware. They show that exploit kits are an integral part of that ecosystem, putting additional emphasis of effective countermeasures. Allodi *et al.* conducted experiments with the server-side code of exploit kits to determine how resilient kits are to changes of targeted systems. In doing so, they found that while some exploit kits aim for a lower, yet steadier infection rate over time,

other kits are designed to deliver a small number of the latest exploits, achieving a higher infection rate [1]. Eshete *et al.* conducted an analysis of the flaws contained in server-side components of exploit kits, showing that half of the investigated back-ends contained exploitable vulnerabilities [10]. Our focus is on more readily observable client-side components of EKs.

Bilge *et al.* show that exploits, which were later on also used in exploit kits, could be found in the wild as zero-days before the disclosure of the targeted vulnerability by the vendor [3]. They show that even with AV that can react to known threats, the window of exposure to zero-days is often longer than expected. Dan Guido presents a case study highlighting the fact that exploit kits encountered by his customers typically incorporated exploits from whitehats or APTs, rather than using a zero-day [14].

Drive-by attacks: Drive-by downloads or drive-by attack have received much attention [5, 12, 24, 29]. Many studies [29, 30, 35, 41] rely on a combination of high- and low-interaction client honeypots to visit a large number of sites, detecting suspicious behavior in environment state after being compromised. Below we mention some of the more closely related projects. Apart from the work that specifically investigates issues related to exploit kits, researchers have also focused on drive-by downloads. In Zozzle, Curtsinger *et al.* develop a solution that can detect JavaScript malware in the browser using static analysis techniques [7]. This system uses a naïve Bayes classifier to finding instances of known, malicious JavaScript. As we have shown in this paper, exploit kits are changing rapidly. Thus, continuously finding suitable training samples for Zozzle is a challenge; additionally, Zozzle did not generate AV signatures. A similar approach was followed by Rieck *et al.* for CUJO, in which the detection and prevention components were deployed in a proxy rather than the browser [33]. Cova *et al.* describe JSAND [6] for analyzing and classifying web content based on static and dynamic features. Their system provides a framework to emulate JavaScript code and determine characteristics that are typically found in malicious code. Ratanaworabhan *et al.* describe Nozzle, a dynamic system that uses a global heap health metric to detect heap-spraying, a common technique used in modern browser exploits [32].

Researchers have also analyzed ways to mitigate the effects of drive-by download attacks. Egele *et al.* check strings that are allocated during runtime for patterns that resemble shellcode and ensure that this code is never executed [9]. Lu *et al.* propose a system called BLADE, to ensure that all executable files that are downloaded via the browser are automatically sandboxed such that they can only be executed with explicit user consent [22]. Kapravelos *et al.* present Revolver, a system that leverages the fact that to avoid detection by emulators or honey clients, authors of exploits use small syntactic changes to throw of such detection tools. To find such evasive malware, they compare the structure of two pieces of JavaScript, allowing them to determine these minor changes [16].

Signature generation: Automated signature generation has been used to counter both network-based attacks and generate AV-like signatures for malicious files. Singh *et al.* proposed an automated method to detect previously unknown worms based on traffic characteristics and subsequently create content-based signatures [36]. Two other research groups present similar works, generating signatures from honeypot [20] and DMZ traffic [18]. Additional research has focused on improving false positive rates of such systems [26], enabling privacy-preserving exchange of signatures to quickly combat detected attacks [40] and shifting the detection towards commonalities between attacks against a single vulnerable service [21, 25].

Work by Brumley *et al.* proposes a deeper analysis of the vulnerabilities rather than exploits to detect malicious packets and subsequently create matching signatures [4]. The concept of clustering HTTP traffic was then used in 2010 by Perdisci *et al.* to find similar patterns in different packets to improve the quality of generated signatures [28]. While much focus has been on the detection of network-based attacks, research into automatic generation of virus signatures dates back to 1994, when Kephart and Arnold propose a system that leverages a large base of benign software to infer which byte sequences in malicious binaries are unlikely to cause false positives if used a signature [17]. In recent years, this idea was picked up when Griffin *et al.* presented Hancock, which determines the probability that an arbitrary byte sequence occurs in a random file and improves the selection of signature candidates by automatically identifying library code in malicious files. This allows them to achieve a false positive rate of 0.1% [13]. FIRMA clusters unlabeled malware based on captured network traffic, using them to produce network signatures with a high precision and recall [31].

In summary, although previous research has been conducted both in EK detection and analysis, as well as the automated generation of signatures, no work has made the connection between the two fields. In contrast, Kizzle leverages insights gathered by us as well as previous work to tailor-make a solution which is allows fully automated detection of Exploit Kits.

VII. Conclusions

This paper proposes Kizzle, a malware signature compiler that targets exploit kits. Kizzle automatically identifies malware clusters as they evolve over time and produces signatures that can be applied to detect malware with a lower false negative and similar false positive rates, when compared to hand-written anti-virus signatures.

While we have seen a great deal of consolidation in the space of Web malware, which leads to sophisticated attacks being accessible to a broad range of attackers, we believe that Kizzle can tip the balance in favor of the defender as it lowers the required effort for the defender,

while simultaneously increasing the work load for the attacker. Kizzle is designed to run in the cloud and scale to large volumes of streaming code samples, allowing for a quick response time to changes in the detected EKs.

Our longitudinal evaluation shows that Kizzle produces signatures of high accuracy. When evaluated over a four-week period in August 2014, false positive rates for Kizzle are under 0.03%, while the false negative rates are under 5%, rivalling manually-crafted AV signatures in both categories.

Acknowledgements

We greatly appreciate the cooperation and help we received from Dennis Batchelder, Edgardo Diaz, Jonathon Green, and Scott Molenkamp in the course of working on this project. This work was partially supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

References

[1] L. Allodi, V. Kotov, and F. Massacci. Malwarelab: Experimentation with cybercrime attack tools. In *Workshop on Cyber Security Experimentation and Test*, 2013.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987.

[3] L. Bilge and T. Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *CCS*, 2012.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE S&P*, 2006.

[5] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security*, 2011.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *WWW*, 2010.

[7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. In *USENIX Security*, 2011.

[8] G. De Maio, A. Kapravelos, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Pexy: The other side of exploit kits. In *DIMVA*, 2014.

[9] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2009.

[10] B. Eshete, A. Alhuzali, M. Monshizadeh, P. Porras, V. Venkatakrishnan, and V. Yegneswaran. Ekhunter: A counter-offensive toolkit for exploit kit infiltration. In *NDSS*, 2015.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *International Conference on Knowledge Discovery and Data Mining*, 1996.

[12] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *CCS*, 2012.

[13] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *RAID*, 2009.

[14] D. Guido. A case study of intelligence-driven defense. *IEEE Security and Privacy*, 2011.

[15] J. Jones. The state of Web exploit kits. In *BlackHat*, 2012.

[16] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security*, 2013.

[17] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Virus Bulletin International Conference*, 1994.

[18] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security*, 2004.

[19] V. Kotov and F. Massacci. Anatomy of exploit kits: Preliminary analysis of exploit kits as software artefacts. In *International Conference on Engineering Secure Software and Systems*, 2013.

[20] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *Workshop on Hot Topics in Networks*, 2003.

[21] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE S&P*, 2006.

[22] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *CCS*, 2010.

[23] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, Jan. 1997.

[24] A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting. In *DIMVA*, Berlin, Germany, July 2013.

[25] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE S&P*, 2005.

[26] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[27] M. Payer. Embracing the new threat: Towards automatically self-diversifying malware. In *The Symposium on Security for Asia Network*, 2014.

[28] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX NSDI*, 2010.

[29] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *USENIX Security*, 2008.

[30] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Workshop on Hot Topics in Understanding Botnets*, 2007.

[31] M. Z. Rafique and J. Caballero. FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors. In *RAID*, St. Lucia, October 2013.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security*, 2009.

[33] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC*, 2010.

[34] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *International Conference on Management of Data*, 2003.

[35] C. Seifert, V. Delwadia, P. Komisarczuk, D. Stirling, and I. Welch. Measurement study on malicious web servers in the .nz domain. In *Australasian Conference on Information Security and Privacy*, 2009.

[36] S. Singh, C. Estan, G. Varghese, and S. Savage. Earlybird system for real-time detection of unknown worms. Technical report, 2003.

[37] SpiderLabs Blog. Rig exploit kit - diving deeper into the infrastructure. online, `https://goo.gl/Ke8t8K`.

[38] B. Stock, B. Livshits, and B. Zorn. Kizzle: A signature compiler for exploit kits. Technical Report MSR-TR-2015-12, February 2015.

[39] TrendLabs Security Intelligence. Exploit kits in 2015: Scale and distribution. online, `http://goo.gl/SlU5sA`.

[40] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *RAID*, 2006.

[41] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. *Managing Information Risk and the Economics of Security*, 2008.

[42] ZScaler ThreatLab. Malvertising, exploit kits, clickfraud and ransomware: A thriving underground economy. `http://goo.gl/ozDmZX`, 2015.