PExy: The other side of Exploit Kits

Giancarlo De Maio*, Alexandros Kapravelos†, Yan Shoshitaishvili†, Christopher Kruegel†, and Giovanni Vigna†

University of Salerno*
demaio@dia.unisa.it
UC Santa Barbara†
{kapravel,yans,chris,vigna}@cs.ucsb.edu

Abstract. The drive-by download scene has changed dramatically in the last few years. What was a disorganized ad-hoc generation of malicious pages by individuals has evolved into sophisticated, easily extensible frameworks that incorporate multiple exploits at the same time and are highly configurable. We are now dealing with *exploit kits*.
In this paper we focus on the server-side part of drive-by downloads by automatically analyzing the source code of multiple exploit kits. We discover through static analysis what checks exploit-kit authors perform on the server to decide which exploit is served to which client and we automatically generate the configurations to extract all possible exploits from every exploit kit. We also examine the source code of exploit kits and look for interesting coding practices, their detection mitigation techniques, the similarities between them and the rise of *Exploit-as-a-Service* through a highly customizable design. Our results indicate that even with a perfect drive-by download analyzer it is not trivial to trigger the expected behavior from an exploit kit so that it is classified appropriately as malicious.

1 Introduction

Over the last few years, the web has grown to be the primary vector for the spread of malware. The attacks that spread malware are carried out by cybercriminals by exploiting security vulnerabilities in web browsers and web browser plugins. Once a vulnerability is exploited, a traditional piece of malware is loaded onto the victims' computer in a process known as a drive-by download [6, 15].

To avoid duplication of effort, and make it easier to adapt their attacks to exploit new vulnerabilities as they are found, attackers have invented the concept of "exploit kits" [1]. These exploit kits comprise decision-making code that facilitates fingerprinting (the determination of what browser, browser version, and browser plugins a victim is running), determines which of the kit's available exploits are applicable to the victim, and launches the proper exploit. As new exploits are developed, they can be added to such kits via a standard interface. Exploit kits can be deployed easily, with no advanced exploitation knowledge required, and victims can be directed to them through a malicious redirect or simply via a hyperlink.

In general, exploit kits fingerprint the client in one of two ways. If the versions of the browser plugins are not important, an exploit kit will determine which of its exploits should be sent by looking at the victim's User-Agent (set by the browser) or the URL query string (set by the attacker when linking or redirecting the user to the exploit kit). Alternatively, if the exploit kit needs to know the browser plugins, or wishes to do some in-depth fingerprinting in an attempt to evade deception, it sends a piece of JavaScript that fingerprints the browser, detects the browser versions, and then requests exploits from the exploit kit, typically by doing a standard HTTP request with a URL query string specifying the victim's detected information, thus reducing this to the first fingerprinting case.

Because of the raw number of different vulnerabilities and drive-by download attacks, and the high rate of addition of new exploits and changes of the exploit kits, the fight against web-distributed malware is mostly carried out by automated analysis systems, called "honeyclients", that visit a web page suspected of malicious behavior and analyze the behavior of the page to determine its maliciousness [12, 16, 5, 14, 17, 10]. These systems fall into two main categories: low-interaction honeyclients and high-interaction honeyclients. The former are systems that heavily instrument a custom-implemented web client and perform various dynamic and static analyses on the retrieved web page to make their determination. On the other hand, the latter are instrumented virtual machines of full systems, with standard web browsers, that are directed to display the given page. When a malicious page infects the honeyclient, the instrumentation software detects signs of this exploitation (i.e., newly spawned processes, network connections, created files, and so on) and thus detects the attack.

In the basic operation of modern honeyclients, the honeyclient visits a page once, detects an exploit, and marks the page as malicious. This page can then be included in a blacklist so that users are protected from being exploited by that specific page in the future. Upon the completion of this process, the honeyclient typically moves on to the next page to be checked.

However, this design represents a humongous missed opportunity for the honeyclients. An exploit kit that is detected in this manner is typically detected based on a single launched exploit. However, in practice, these exploits hold anywhere up to a dozen exploits, made for many different browsers and different browser versions. We feel that simply retrieving a single exploit and detecting the maliciousness of a page is not going far enough: every additional exploit that can be retrieved from the exploit kit provides additional information that the developers of honeyclients can use to their advantage.

For example, it is possible for honeyclients and other analysis systems to use signatures for quicker and easier detection. A high-interaction honeyclient can create a signature from the effects that a certain exploit has on the system, and this signature could be used by both the honeyclient itself and by other attack-prevention systems (such as antivirus systems) to detect such an exploit in the future. Similarly, low-interaction honeyclients can create signatures based on the contents of the exploit itself and the setup code (typically very specific

techniques, such as heap spraying, implemented in JavaScript). These signatures could then be passed to a similarity-detection engine, such as Revolver [8], which can detect future occurrences of this exploit. Finally, an opportunity is missed when moving on from an exploit kit after analyzing only one exploit because other, possibly high-profile, exploits that such a kit might possess will go ignored. If one of these exploits is previously unseen in the wild (i.e, it is a 0-day), detecting it as soon as possible is important in minimizing the amount of damage that a 0-day could cause.

Our intuition is that, by statically analyzing the server-side source code of an exploit kit (for example, after the server hosting it has been confiscated by the authorities and the kit's source code has been provided to the researchers), a set of user agents and query string parameters can be retrieved that, when used by a honeyclient, will maximize the number of exploits that can be successfully retrieved. Additionally, because exploit kits share similarity among family lines, these user agents and query string parameters can be used to retrieve exploits from other, related exploit kits, even when the server-side source code of these kits is not available. By leveraging these intuitions, it is possible to extract a high amount of exploits from these exploit kits for use in similarity detection, signature generation, and exploit analysis.

To demonstrate this, we designed a system called PExy, that, given the source code of an exploit kit, can extract the set of URL parameters and user agents that can be combined to "milk" an exploit kit of its exploits. Due to the way in which many of these kits handle victim fingerprinting, PExy frequently allows us to completely bypass the fingerprinting code of an exploit kit, even in the presence of adversarial fingerprinting techniques, by determining the input (URL parameters) that the fingerprinting routine would provide to the exploit kit. We evaluate our system against a collection of over 50 exploit kits in 37 families by showing that it can generate the inputs necessary to retrieve 279 exploits (including variants).

This paper makes the following contributions:

- We provide an in-depth analysis of a wide range of exploit kits, using this to motivate the need for an automated analysis system.
- We present the design of a framework for static analysis of exploit kits, focusing on the inputs that those kits process during their operations.
- We develop and demonstrate a technique to recover the necessary inputs to retrieve a majority of an exploit kit's potential output, focusing on retrieving as many exploits from exploit kits as possible.

2 Anatomy of an Exploit Kit

In this section, we will detail the anatomy of exploit kits, derived from a manual examination of over 50 exploit kits from 37 different families (detailed in Figure 3, to help the reader understand our decisions in developing the automated approach.

In general, the lifecycle of a victim's interaction with an exploit kit proceeds through the following steps.

1. First, the attacker lures the victim to the exploit kit's "landing page". This is done, for example, by sending a link to the victim or injecting an IFrame in a compromised web page.
2. The victim's browser requests the exploit kit's landing page. This interaction can proceed in several ways.
   (a) If the exploit kit is capable of client-side fingerprinting, it will send the fingerprinting JavaScript to the client. This code will then redirect the client back to the exploit kit, with the fingerprinting results in URL parameters.
   (b) If the exploit kit is incapable of client-side fingerprinting, or if the request is the result of the client-side fingerprinting code, the exploit kit selects and sends an exploit to the victim.
3. The victim's browser is compromised by the exploit sent by the exploit kit, and the exploit's payload is executed.
4. The exploit payload requests a piece of malware from the exploit kit, downloads it, and executes it on the user's machine. This malware (typically a bot) is generally responsible for ensuring a persistent infection.

2.1 Server-side Code

The analyzed exploit kits in our dataset are web applications written in PHP, and most of them use a MySQL database to store configuration settings and exploitation statistics. We will describe several main parts of these exploit kits: server-side modules (such as administration interfaces, server-side fingerprinting code, and exploit selection), and client-side modules (such as fingerprinting and exploit setup code).

Exploit kit	**Encoding**	**Decoding**
Blackhole 1.1.0	IonCube 6.5	Partial
Blackhole 2.0.0	IonCube 7	Partial
Crimepack 3.1.3	IonCube 6.5	Full
Crimepack 3.1.3-b	IonCube 6.5	Full
Tornado	ZendGuard	Full

Table 1 – Server-Side encoding.

Obfuscation. Some exploit kits are obfuscated with commercial software such as IonCube and ZendGuard (Table 1). It was possible to break the encoding, albeit only partially in some cases, by means of the free service provided at `http://easytoyou.eu` and other tools from the underground scene[1].

[1] See `http://ioncubedecoder2013.blogspot.com/2013/05/ioncube-decoder.html`

Database. Most exploit kits are capable of recording information about victims that are lured to visit them. While some kits (such as the Tornado exploit kit) store this information on the filesystem, most maintain it in a MySQL database. Furthermore, all of the examined samples provide an administrative web interface meant to access and analyze these statistics.

Administration Interface. The exploit kits in our dataset all implement an administrative web interface, with varying degrees of sophistication. This password-protected interface enables the administrator of the exploit kit to configure the exploit kit and view collected victim statistics.

The configurability of exploit kits varies. All of the exploit kits that we analyzed allowed an administrator to upload malware samples that are deployed on the victim's machine after the victim is successfully exploited. More advanced exploit kits allow fine-grained configuration. For example, Blackhole, Fragus, and Tornado allow the creation of multiple instances (termed "threads" by the exploit kits' documentation), each exhibiting a different behavior (typically, different exploits to attempt and malware to deliver). These threads are associated with different classes of victims. For example, an attacker might configure her exploit kit to send different pieces of malware to users in the United States and users in Russia.

2.2 Fingerprinting

All of the exploit kits in our dataset implement a fingerprinting phase in which information about the victim is collected. This information is used by the exploit kit to select the appropriate exploit (according to the type and versions of software running on the victim's computer) and to defend the kit against security researchers. Such information can be collected on either the server or the client side, and can be used by an exploit kit to respond in a different way to different victims.

Fingerprinting results can also be used for evasion. For example, if the victim is not vulnerable to any of the kit's exploits, or the IP address of the victim is that of a known security research lab (or simply not in a country that the attacker is targeting), many exploit kits respond with a benign web page.

Additionally, many exploit kits deny access to the client for a period of time between visits in an attempt to be stealthy. Exploit kits without a server-side database typically implement this by using cookies, while those with a database store this information there.

Server-side Fingerprinting. A request to a web page may carry lot of information about the victim, such as their HTTP headers (i.e., the User-Agent, which describes the victim's OS family and architecture and their browser version), their IP address (which can then be used, along with the Accept-Language header, to determine their geographic location), URL parameters (which can be set by client-side fingerprinting code), cookies (that can help determine if the

client already visited the page) and the HTTP *Referer* header. A typical example of behavioral-switching based on server-side fingerprinting is shown in Listing 1.1, extracted from the Armitage exploit kit, where the choice of the exploit to be delivered depends on the browser of the victim. While in this case, the information was derived from the User-Agent, other exploit kits receive such information in the form of URL parameters from client-side fingerprinting code.

```
if( $type == "Internet Explorer" )
  include("e.php");
if( $type == "Opera" && $bv[2]<"9.20" && $bv[2]>"9" )
  include("opera.php");
if( $type == "Firefox" )
  include("ff.php");
```

Listing 1.1 – Behavior based on the victim's browser (Armitage).

Client-side Fingerprinting. Because client-side fingerprinting can give a more accurate view of the client's machine, most of the exploit kits implement both server-side and client-side fingerprinting. Client-side fingerprinting is used to retrieve information unavailable from HTTP headers, such as the victim's installed browser plugins and their versions. Since many browser vulnerabilities are actually caused by vulnerabilities in such plugins (most commonly, Adobe Reader, Adobe Flash, or Java), this information is very important for the selection of the proper exploit.

```
var a_version = getVersion("Acrobat");
  if(a_version.exists){
    if(a_version.version >= 800 && a_version.version < 821){
      FramesArray.push("load_module.php?e=Adobe-80-2010-0188"
          );
    }else if(a_version.version >= 900 && a_version.version <
        940){
      if(a_version.version < 931){
        FramesArray.push("load_module.php?e=Adobe
            -90-2010-0188");
...
var newDIV=document.createElement("div");
newDIV.innerHTML="<iframe src='" + FramesArray[CurrentModule]
     + "'></iframe>";
document.body.appendChild(newDIV);
```

Listing 1.2 – Requests generated client-side (Bleeding Life v2.0).

The retrieved information is passed back to the exploit kit via an HTTP GET request, with URL parameters denoting the client configuration. An example of how these requests are generated in client-side fingerprinting code is shown in Listing 1.2. The excerpt, extracted from Bleeding Life v2.0, makes use of the PluginDetect library[2] to obtain information about the Adobe Acrobat plugin in

[2] http://www.pinlady.net/PluginDetect/

Internet Explorer. Depending on the plugin version, a subsequent request is constructed to retrieve the proper exploit. Although the fingerprinting is happening on the client side, the server is still the one that is distributing the exploit and makes a server-side decision (based on the URL parameters sent by the client-side fingerprinting code) of which exploit to reveal. Listing 1.3, extracted from the Shaman's Dream exploit kit, shows how the result of a client-side fingerprinting procedure (stored in the "exp" URL parameter) is used on the server-side to select the exploit.

```
...
$case_exp = $_GET["exp"];

if ($browser == "MSIE"){
  if ($vers[2] < "7"){
    if (($os == "Windows XP") or ($os == "Windows 2003")){
      switch ($case_exp) {
        case 1: echo _crypt(mdac()); check();break;
        case 2: echo "<html><body>"._crypt(DirectX_DS7())."</
            body></html>";      check();break;
        case 3: echo _crypt(Snapshot()); check();break;
        case 5: echo _crypt(msie_sx()); check();break;
        case 4: echo _crypt(pdf_ie2()); die;break;
...
```

Listing 1.3 – Execution-control parameters (Shaman's Dream).

2.3 Delivering the Exploits

Exploit kits contain a number of exploits, of which only a subset is sent to the victim. This subset depends on the output of the fingerprinting step, whether the fingerprinting is done only server-side or on both the server and client side. The kits that we have analyzed use the following information to pick an exploit to deliver.

IP headers. The IP address of the victim, stored by PHP as a global variable in `$_SERVER['REMOTE_ADDR']`, is used by exploit kits for geographical filtering. For example, an exploit kit administrator might only want to infect people in the United States.

HTTP headers. HTTP headers, stored by PHP in the `$_SERVER` global array, carry a lot of information about the victim. Exploit kits typically use the following headers:

User-Agent. Exploit kits use the user agent provided by the victim's browser to determine which OS family, OS version, browser family, and browser version the victim's PC is running.

Accept-Language. Along with the IP address, this header is used by exploit kits for geographical filtering.

Referer. This header is used by exploit kits for evasive purposes. Some kits avoid sending malicious traffic to victims when no referrer is present, as this might be an indication of an automated drive-by-download detector.

Cookies. Cookies are used to temporarily "blacklist" a victim from interaction with the exploit kit. They are accessible from PHP via the `$_COOKIE` variable.

HTTP query parameters. Finally, exploit kits use HTTP query parameters (i.e., URL parameters in a GET request or parameters in a POST request) quite heavily. These parameters, accessed in PHP through the `$_QUERY` global variable, are used for two main purposes: receiving results of fingerprinting code, and internal communication between requests to the exploit kits.

Receiving fingerprinting results. Client-side fingerprinting code relays its results back to the exploit kit via URL parameters. As exemplified in Listing 1.3, this information is then used to select the proper exploits to send to the victim.

Inter-page communication. By examining the exploit kits manually we found out that the majority of the analyzed exploit kits (41 out of 52) employ URL parameters to transfer information between multiple requests. In some cases, such as the bomba and CrimePack exploit kits, there were up to 6 parameters used.

2.4 Similarity

Our analysis of the exploit kits revealed that many kits share common code. In fact, the source code is almost identical between some versions of the exploit kits, leading to the conclusion that these kits were either written by the same individual or simply forked by other criminals. Such similarities between exploit kits can be leveraged by security researchers, as effective techniques for analyzing a given kit are likely to be applicable to analyzing related kits.

To explore the implications of these similarities, we analyzed a subset of our dataset using Revolver, a publically available service that tracks similarities of malicious JavaScript [8]. The results, shown in Figure 1, demonstrate the evolution of these exploit kits. We see three main families of exploit kits emerge from the analysis: Blackhole, which contains very characteristic code within its exploit staging, MPack / Ice Pack Platinum / 0x88, which appear to share exploitation scripts, and Eleonore / MyPolySploits / Unknown / Cry / Adpack / G-Pack, which share (albeit slightly modified) exploits as well. Additionally, manual analysis of the back-end PHP code confirmed that these exploit kits use similar code, and are probably derived from each other.

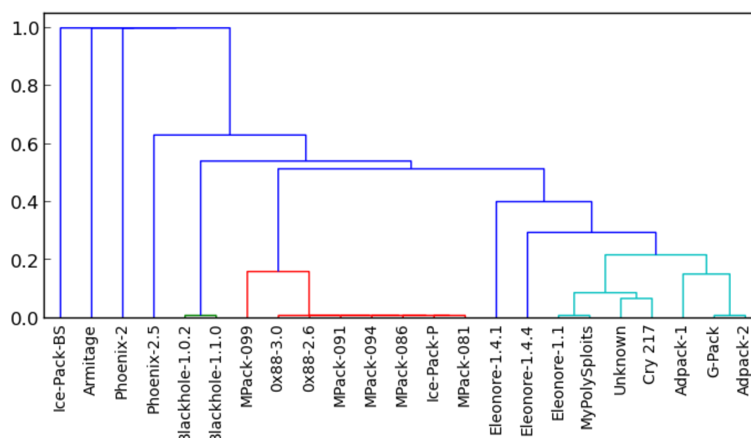


Fig. 1 – Exploit kit similarities identified by *Revolver*. The lower the U-shaped connection, the higher the similarity.

3 Automatic Analysis of Exploit Kits

In this work we propose a method to automatically analyze an exploit kit given its source code. Our objective is to extract the inputs due to which the exploit kit changes its behavior. This can be used by web-malware analyzers to both classify websites correctly and milk as many exploits as possible from exploit-kit deployments found in the wild.

Milking an exploit kit involves the creation of a set of inputs to trigger all the possible behaviors in order to obtain as many exploits as possible, which may improve the analysis of the page. This is a problem of code coverage, with the constraint that only a specific subset of variables can be tuned. The subset of tunable variables is extracted by the PHP engine from the victim's HTTP request.

The source code of an exploit kit may contain several paths depending on HTTP parameters. The challenge is to be able to discern whether a parameter affects the behavior of the exploit kit. An exploit kit may be characterized by a set of behaviors, where each behavior is an execution path that maps a request to a different response.

In essence, this problem can be reduced to (1) identifying all the branches in the code that depend on (tunable) HTTP elements and (2) determining the values of the parameters to satisfy the condition. By doing this, we can obtain, for each exploit kit:

- A list of HTTP elements that characterize the exploit kit.
- A list of values for those elements that can be used to cover as much server-side code as possible.

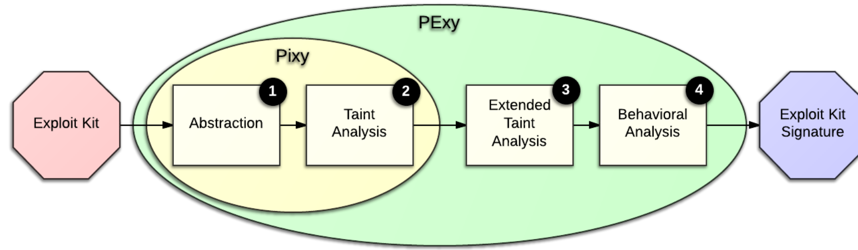


Fig. 2 – Architecture of PExy.

3.1 System Design and Architecture

The main contribution of this work is PExy, a system for the automatic analysis of the source code of exploit kits. The high-level architecture of PExy is presented in Figure 2.

An exploit kit submitted to PExy undergoes a four-stage analysis. In the first place, an abstract representation of the source code, the Control Flow Graph (CFG), is generated (1). The CFG is then processed by a taint analyzer that extracts a first level of information about the HTTP parameters used by the exploit kit (2). These initial steps are accomplished by means of Pixy [7]. However, as we discuss in Section 3.3, the information gathered so far is not sufficient to accomplish an accurate behavioral analysis. In order to extract the missing information, an extended taint analysis is performed (3). This knowledge is then passed to the behavioral analyzer, which is able to discern the HTTP parameters and values that influence the behavior of the exploit kit (4). The output of PExy is a signature of the exploit kit that can be used by a honeyclient to both identify and milk similar exploit kits in the wild.

PExy inherits most of data structures defined by Pixy. For sake of clarity, a brief overview of Pixy is presented below.

3.2 Pixy: Data-Flow Analysis for PHP

Pixy is a flow-sensitive, interprocedural, and context-sensitive data flow analysis system for PHP, targeted at detecting taint-style vulnerabilities. It is also available as a fully-fledged prototype implementing the proposed analysis technique.

The first phase of the analysis consists of generating an abstract syntax tree representation of the input PHP program, which is the Parse Tree. The Parse Tree is then transformed into a linearized form resembling Three-Address Code (TAC). At this point, a Control Flow Graph (CFG) for each encountered function is constructed.

In order to improve correctness and precision of the taint analysis, the methodology includes two further phases: alias and literal analysis. It is worth noting that, whenever a variable is assigned a tainted value, this taint value should not

be only propagated to the variable itself, but also to all its aliases (variables pointing to the same memory location). In order to handle this case, an alias analysis to provide information about alias relationships is performed.

On the other hand, literal analysis is accomplished in order to deduce, whenever possible, literal values that variables and constants may hold at each program point. This information is used to evaluate branch conditions and ignore program paths that cannot be executed at runtime.

The analysis technique is aimed at detecting taint-style vulnerabilities, such as XSS, SQL injection and command injection flaws. In this context, tainted data can be defined as data that originates from potentially malicious users and can cause security problems at vulnerable points in the program. In order to accomplish this task, three main elements are defined by Pixy:

1. *Entry Points* - any elements in the PHP program that can be controlled by the user, such as HTTP POST parameters, URL queries and HTTP headers;
2. *Sensitive Sinks* - all the routines that return data to the browser, such as `echo()`, `print()` and `printf()`;
3. *Sanitization Routines* - routines that destroy potentially malicious characters, such as `htmlentities()` and `htmlspecialchars()`, or type casts that transform them into harmless ones (e.g., casts to integer).

The taint analysis implemented by Pixy works as follows. First, the Sensitive Sinks of the program are detected. Then, for each Sensitive Sink, information from the data-flow analysis is used to construct an acyclic dependency graph for its input. A vulnerability is detected if the dependency graph contains a path from an Entry Point to the Sensitive Sink, and no Sanitization Routines are performed along this path.

3.3 PExy: Static Analysis of Malicious PHP

The main goal of PExy is to perform the behavioral analysis of a PHP code, aimed to determine which HTTP parameters and values influence the execution of an exploit kit. To accomplish this task, we enriched the taint analysis implemented by Pixy with new techniques that allow us to classify all the branches in the input program. The information extracted by means of this *extended* taint analysis is used to discern the behavior of the exploit kit. The behavioral analysis is further divided into different sub-phases, each based on a targeted set of heuristics. In detail, PExy performs the following activities:

1. First-level taint analysis (Pixy)
2. Branch identification through extended taint analysis
3. Branch classification
4. Parameter and value extraction
5. Value determination

First-level taint analysis. The first activity performed by PExy is the identification of all the branches in the program that depend on client's parameters. This can be accomplished by tainting the corresponding elements in the PHP program (i.e., `$_GET`, `$_POST`, `$_QUERY`, `$_SERVER`, `$_COOKIE` arrays), which have been previously defined as Pixy *Entry Points*. The main difference is that we are now interested in how these parameters influence the behavior of a malicious script. To understand this, we configured PExy to treat all conditions as Pixy Sensitive Sinks. A Sensitive Sink corresponding to a conditional branch is referred as Condition Sink.

The output of Pixy is a set of all the Condition Sink encountered in the program with relative taint information (dependence graphs).

Branch identification through extended taint analysis. Any missed values from tainting would greatly impact PExy's precision, and so it is important to support *indirect taint*. In Pixy's normal operation, a tainted value may be passed from a variable `X` to another variable `Y` if the value of `X` is transferred to `Y` as result of some operations. In the context of our analysis, however, this definition is too restrictive and needs to be expanded with new rules. Consider the example in Listing 1.4. In such a case, it is clear that the second condition is *indirectly* dependent on the tainted variable `$_GET['a']`, since the value of `$a` is part of a control-flow path that is depending on `$_GET['a']`.

```
if( $_GET['a']=='1' ){
  # the taint should be transfered to $a
  $a='doit';
}
...
# this indirectly depends on $_GET['a']
if( $a=='doit' ){
  echo($exploit1);
}
```

Listing 1.4 – Example of indirect tainting

In order to handle these cases, we leverage the concept of *indirect taint*. An indirect taint is transferred from a variable `X` to a variable `Y` if the value of `Y` *depends* on `X`. Clearly, this rule is more general since it does not imply that `Y` contains the same data of `X`. This new definition allows handling cases as that shown before: if `X` is tainted and `Y` is updated depending on the value of `X`, then `Y` will be tainted in turn. In order to implement indirect tainting, we extended the taint analysis implemented by Pixy accordingly.

A further analysis of the dependence graphs generated by Pixy allows to discern indirect dependences among the Condition Sinks. The dependence graph of each Condition Sink is eventually augmented with this information.

After identifying all the conditions depending on client parameters, we can perform a reduction step. Because of the TAC representation, the expression of a condition is split in a series of simpler binary conditions. Therefore, a single

condition in the original code may determine multiple conditions in the CFG. Splitting conditions like this allows us to isolate tainted inputs from other system conditions and reduce the complexity of future steps in the analysis.

The output of this phase is a set of Condition Sinks (and relative taint information) whose outcome in the original code is determined by one or more request parameters.

Branch classification. The previous step yields the list of all conditional branches of the exploit kit that depend on client parameters. We then aim to discern how these parameters influence the behavior of the program. We define a change of behavior as a change in the response to be sent to the client, which depends on one or more request parameters.

In order to identify these cases, we detect *Behavioral Elements* in the program. A Behavioral Element is defined as an instruction, or block of instructions, that manipulate the server's response. In particular, we are interested in Behavioral Elements depending on Condition Sinks. We have identified four distinct classes of Behavioral Elements: embedded PHP code, print statements, file inclusion, and header manipulation.

Embedded PHP code. One method with which an attacker can generate a response to the victim is via the use of embedded PHP code, allowing the kit to interleave HTML code with dynamic content computed server-side at runtime.

Printing statements. Print functions, such as `echo()` and `print()`, are often used by exploit kits to manipulate the content of the response. We use the data-flow analysis algorithm of Pixy to identify these elements. In addition, we analyze the dependency graphs of these elements in order to retrieve information about the output.

File inclusion. PHP allows dynamic code inclusion by means of built-in functions such as `include()` and `readfile()`. In our analysis, we found that most of the kits use dynamic file inclusion to load external resources. Thanks to the literal analysis implemented by Pixy, it is possible to reconstruct the location of the resource and retrieve its content. The content of the resource is then analyzed by taking its context in the program into account.

Header manipulation. HTTP headers are typically manipulated by exploit kits to redirect the client to another URL (by setting the `Location` header) or to include binary data, such as images, in the response (by modifying the MIME type of the body of the request by means of the `Content-Type` header). In order to detect these cases, we analyze the calls to the `header()` function and try to reconstruct the value of its argument. If the call sets a Location or Content-type header, we add the position to the list of the Behavioral Elements.

Once we have obtained the possible Behavioral Elements of the program, we add all conditional branches upon which a Behavioral Element depends to a list of Behavioral Branches, which is the output of this phase.

Parameter and value extraction. PExy next determines, for each Behavioral Branch, the type, name and value of the HTTP request parameter that satisfies the branch condition. It can be accomplished by analyzing the dependency graphs of the branch condition.

It is worth recalling that, due to the TAC conversion, each complex condition of the program has been split in multiple binary conditions. By leveraging this fact, we can extract a subgraph of operand dependencies from the dependency graph, and focus our analysis on the tainted parameters and the values against which they are compared by the branch condition. If the comparison value is hard-coded in the source code (e.g., a literal), and not computed at runtime (e.g., as result of a database query), it is possible to determine the constraints imposed upon the tainted parameter itself by the branch condition.

Value determination. The next step of our analysis is the determination of the value that a given HTTP parameter must have to satisfy a condition. Typical operations used in condition statements are binary comparison operations like: `===`, `==`, `!=`, `<`, `>`, `<=`, `>=` or the unary `!` operation. We also address some common cases in which the operation is not a comparison, but a call to a built-in function that returns a boolean value. Some examples are the `isset()` and `strstr()`, which are largely used by exploit kits to check values of client's parameters.

By analyzing the branch condition constraints, we are able to retrieve the required string contents of our tainted HTTP parameters.

Indirect Tainted Variables. In most of cases, the condition depending on the user-agent string is performed against an indirectly-tainted variable. As consequence, the value of the variable does not contain any information about the original parameter. A real-word example of this situation is given in Listing 1.5, extracted from the Ice-Pack exploit kit. In that case, the value 1 is referred to Internet Explorer. The value that contains the semantically meaningful information is in the condition where the current value (1) is assigned to the indirect-tainted variable. Thanks to the indirect tainting algorithm, we know the original Behavioral Branch based on which indirect tainted value is updated. By propagating branch conditions through indirectly tainted variables, we are able to reconstruct the indirect tainting dependences.

```
if(strpos($agent, 'MSIE') ){
  $browers=1;
  ...
}
else if ( strstr( $agent, "Opera" ) ){
  $browers=2;
  ...
}
...
if ( $browers == 1 ){
  if ( $config['spl1'] == 'on' && $vers[0] < 7 ){
```

```
    include("exploits/x1.php");
  }
  ...
}
```

Listing 1.5 – Indirect browser selection in Ice-Pack v3

4 PExy: Analysis Results

PExy has been tested against all the exploit kits shown in Figure 3 except for the Blackhole family, which was compiled to a binary. A total of more than 50 exploit kits and 37 different families were analyzed and 279 exploit instances were found. A deeper insight of the characteristics of these samples is given in Section 2. For our results, we consider a false negative a condition leading to a change in exploit-kit behavior that is not correctly classified by PExy as Behavioral Branch. On the other hand, a false positive is a Behavioral Branch that does not lead to a change in exploit-kit behavior.

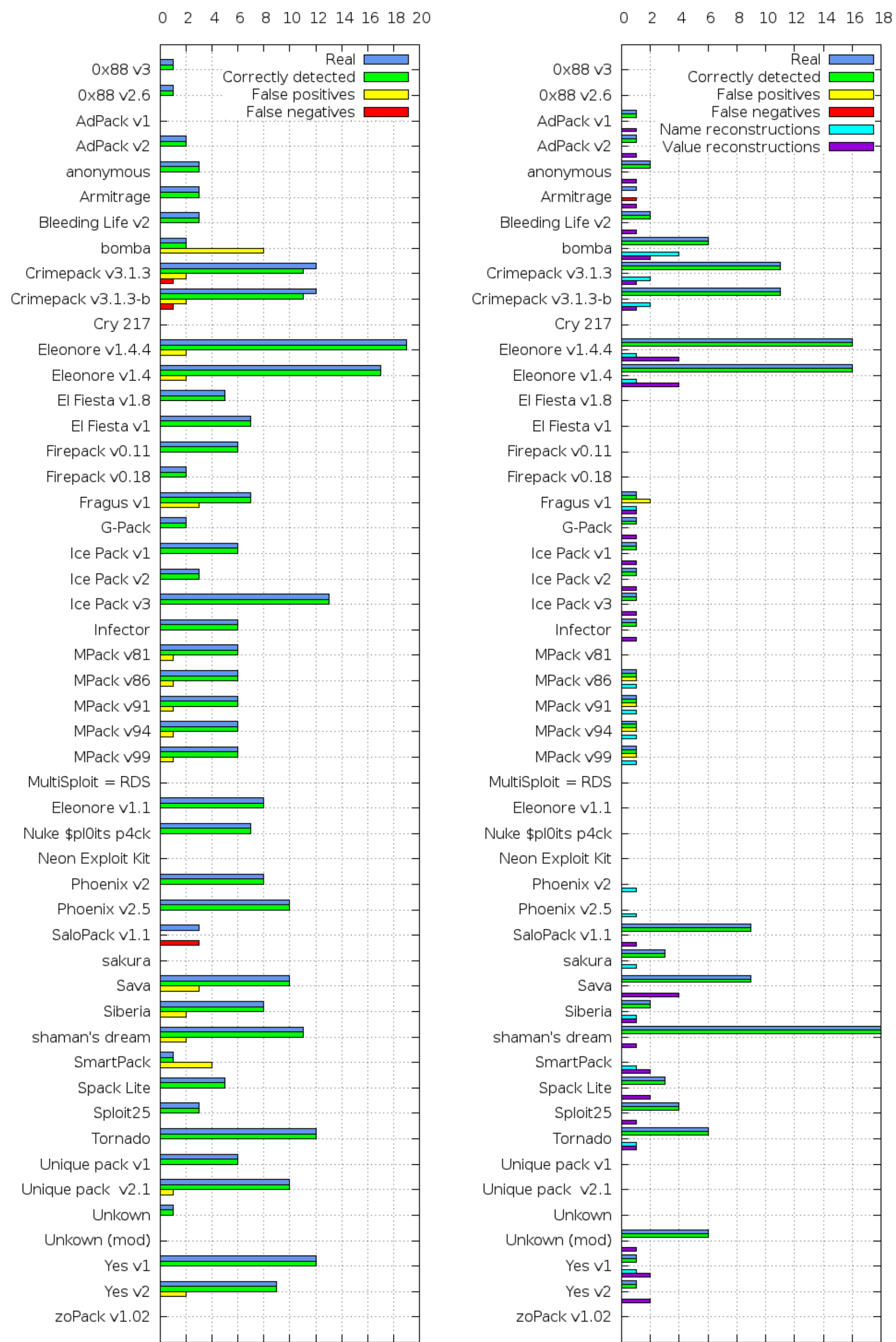
4.1 User-Agent analysis.

In all the cases listed in Figure 3, PExy has been able to identify all the conditional branches depending on the User-Agent value. The branch classification produced few false positives (conditions that do not lead to distinct output) and just one case with false negatives (undetected conditions). A summary of these result is shown in Figure 3a. In all the cases, PExy has been able to reconstruct the proper User-Agent header.

The false negatives in the case of SaloPack are due to the fact that the branches depending on the User-Agent are in a function called by means of the SAJAX toolkit[3]. This library invokes PHP functions from JavaScript by transparently using AJAX. Analyzing this would require to interpret the client-side code. Client-side JavaScript analysis, however, is out of the scope of this work. The fact that only one kit from our dataset exhibited such behavior shows that, in almost all cases, the pertinent HTTP parameters can be extracted from purely server-side analysis.

In Table 2 we show the most and least popular User-Agents that PExy detected in the analyzed exploit kits. One of the most vulnerable configurations that we found with PExy is Internet Explorer 6. There have been more than 100 vulnerabilities for Internet Explorer 6 and the fact that it is usually deployed on a Windows XP machine makes it an easy target for the attackers. It is quite surprising that many exploit kits have an exploit for the Opera browser. This is very hard to detect with honeyclients, as it is a configuration that it is not very popular. In the least favorite exploits we found that the targeted configurations include the Konqueror browser and other Internet Explorer versions that are not widely deployed (versions 5.5 and 7.0).

[3] http://www.modernmethod.com/sajax/



(a) Malicious paths depending on the User-Agent header

(b) Malicious paths depending on GET parameters

Fig. 3 – Summary of the information extracted from the Exploit Kits

# exploit kits	User-Agent
39	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
35	Mozilla/3.0 (compatible)
15	Opera/9.0 (Windows NT 5.1; U; en) Opera/9.0 [...]
1	Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)
1	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; [...]
1	Mozilla/5.0 (compatible; Konqueror/3.4; Linux 2.6.8; [...]

Table 2 – The top three and bottom three User-Agents used by exploit kits to determine which exploit to serve.

4.2 Parameter analysis.

As in the previous case, all the conditions depending on URL parameters have been correctly detected by PExy. As reported in Figure 3b, there are 5 false positives (misclassification) and one case with false negatives (misdetection).

In many cases, PExy has been able to correctly reconstruct not only the correct name of the parameter, but also all of its possible values leading to different code paths, as shown in Figure 3b.

5 Applications

The output of PExy is a signature that includes supported User-Agents and HTTP parameters used by the exploit kit. This information can be used in both an active and a passive way.

Milking. Given a URL pointing to an exploit kit that belongs to a known family, the signature generated by PExy can be used to forge a number of requests able to milk the malicious content. It is worth noting that the information extracted by PExy allows forging a set of requests that are targeted to the specific exploit-kit family. Without this information, the analyzer should perform at least one request for each potentially targeted browser (brute-force approach), which if done blindly can lead to thousands of additional requests [2]. Considering that the number of different browsers that PExy detected in the analyzed exploit kits is 25, the analyzer enhanced with the results of PExy should perform at most 25 requests to milk a malicious website.

The information produced by PExy may noticeably reduce the overall number of requests to be generated by the analyzer, as shown in Table 3. This result is even more important considering that each request for the same page should be generated from a different IP address to avoid blacklisting.

Prior to this work, a drive-by download detection system would stop after getting a malicious payload from a website. With PExy we show that it is possible to leverage our knowledge on exploit kits and reveal more exploit code. This can be highly beneficial to analysis systems, by expanding their detected exploits and pinpointing false negatives.

Honeyclient setup. Another important application of PExy is the vulnerable browser configuration setup. With PExy we are able to determine the exact settings of a browser, such as its version and its plugins, so that we trigger different exploits when visiting an exploit kit. This information is very important to drive-by analyzers, which if they are not configured properly will never receive a malicious payload from the visited exploit kit. PExy not only limits the possible vulnerable browser configurations, but can also provide the least amount of configurations to trigger an exploit in all analyzed exploit kits.

Total unique User-Agents	25
Maximum User-Agents per EK	9
Average User-Agents per EK	3.38

Table 3 – Advantage on using PExy over brute-force.

6 Limitations

With PExy we study the server-side component of exploit kits, but there is also client-side code involved in a drive-by download. Instead of focusing on how the client-side code is fingerprinting the browser, we study the server-side implications of the fingerprinting. This way we will miss how the URL parameters get generated from JavaScript, but we will see how they affect the exploit-kit's execution flow.

A fundamental limitation of PExy is the availability of exploit-kits' server-side source code. With the attackers moving to an Exploit-as-a-Service model of providing exploit kits, the only legal way to obtain the source code is with law enforcement takedowns. This is forcing the security researchers to treat so far exploit kits as a black box. Although PExy was applied in a subset of exploit kits, we believe that the results can help researchers understand exploit kits in a better way.

7 Related Work

Exploit kits have become the de facto medium to deliver drive-by downloads. There have been many techniques proposed to detect drive-by downloads. Cova *et al.* [5] proposed an emulation based execution of webpages to extract the behavior of JavaScript code and the use of machine-learning techniques to differentiate anomalous samples. An attack-agnostic approach was introduced in BLADE [11] based on the intuition that unconsented browser downloads should be isolated and not executed. Our work differs in that we study the server side component of exploit kits and not the drive-by downloads that are served.

The Exploit-as-a-Service model for compromising the browser has been studied by Grier et al. [4]. Their work differs in that they focus on the malicious

binary delivered after the infection, while we focus on the server-side code that delivers the exploit.

Fingerprinting the browser is an important step in the exploitation process. Recent work has shown how this is done as part of commercial websites to track users and for fraud detection [13, 3]. This is different from how the exploit kits fingerprint the browser, since they are not trying to create a unique ID of the browser but determine its exact configuration.

Kotov *et al.* [9] have conducted a preliminary manual analysis of exploit-kits' source code describing their capabilities. We focus on understanding how the client side configuration of the browser affects the server side execution of exploit kits and how it is possible to extract the most exploits out of an exploit-kit installation automatically.

8 Conclusion

In this paper we give a deep insight into how exploit kits operate to deliver their exploits. We build a static analysis system called PExy that is able to analyze an exploit kit and provide all the necessary conditions to trigger all exploits from an exploit kit. We show that we can detect automatically all the paths to malicious output from exploit kits with very few false negatives and false positives. This information can be valuable to drive-by download analyzers, expanding their detections to additional exploits. Even the most accurate drive-by download analyzer needs to be configured with the right browser version and plugins to be exploited. PExy can give the exact set of configurations that an analyzer needs to be as exploitable as possible.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, and Secure Business Austria.

References

1. A criminal perspective on exploit packs. http://www.team-cymru.com/ReadingRoom/Whitepapers/2011/Criminal-Perspective-On-Exploit-Packs.pdf.
2. UA Tracker statistics. http://www.ua-tracker.com/stats.php.
3. G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. Fpdetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.
4. C. Grier et al. Manufacturing compromise: The emergence of exploit-as-a-service. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, October 2012.

5. M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.
6. C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2012.
7. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.
8. 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.
9. V. Kotov and F. Massacci. Anatomy of exploit kits. In *Engineering Secure Software and Systems*, pages 181–196. Springer, 2013.
10. L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2010.
11. L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 440–450. ACM, 2010.
12. J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.
13. N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.
14. N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proc. of the USENIX Security Symposium*, 2008.
15. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proc. of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.
16. P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proc. of the USENIX Security Symposium*, 2009.
17. Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2006.