MalwareLab: Experimentation with Cybercrime Attack Tools

Luca Allodi
DISI - University of Trento.
Via Sommarive 5, Povo (TN), Italy

Vadim Kotov
DISI - University of Trento.
Via Sommarive 5, Povo (TN), Italy

Fabio Massacci
DISI - University of Trento.
Via Sommarive 5, Povo (TN), Italy

Abstract

Cybercrime attack tools (i.e. Exploit Kits) are reportedly responsible for the majority of attacks affecting home users. Exploit kits are traded in the black markets at different prices and advertising different capabilities and functionalities. In this paper we present our experimental approach in testing 10 exploit kits leaked from the markets that we deployed in an isolated environment, our *MalwareLab*. The purpose of this experiment is to test these tools in terms of resiliency against changing software configurations in time. We present our experiment design and implementation, discuss challenges, lesson learned and open problems, and present a preliminary analysis of the results.

1 Introduction

In the cybercrime underground markets attack tools and software artefacts are constantly traded [3, 10]; these are responsible, reportedly, for about two thirds of user infections worldwide [8]. A class of these tools, namely "Exploit Kits", seem to be particularly popular among cyber-criminals, to the point that both industry [10] and academia [3] got recently interested in the phenomenon. An exploit kit is a software tool used by cyber criminals to deliver drive-by-download attacks. It is an HTTP server-side application, that, based on request headers, returns a page with an appropriate set of exploits to the victim computer. Exploit kits are traded in the black markets and explicitly advertise the vulnerabilities they exploit. We performed a systematic exploration of underground and public channels and gathered more than 30 exploit kits, spanning from 2007 to 2011.

In this paper we describe our experimental approach to test exploit kits in terms of resiliency in time and namely for how long an exploit kit would work considering the pace of software evolution. We conduct our experiments in an isolated environment, the MalwareLab, built for this purpose and maintained at the University of Trento, Italy. We discuss our design and implementation methodology, and present the results of our analysis. We also discuss strengths and weaknesses of our design, and potential flaws of our implementation.

Section 2 and 3 give a brief background on exploit kits and discuss related work respectively. In Section 4 we report a first (failed) experiment design. The paper then proceeds with describing a second experiment design (Section 5) and implementation (Section 6). We then discuss in Section 7 preliminary results of the experiment. Section 8 presents open points and challenges we identify in our design and implementation, and Section 9 concludes the paper.

2 Background on exploit kits

Exploit kits' main purpose is to silently download and execute malware on the victim machine by taking advantage of browser or plugin vulnerabilities. Errors in applied programming interfaces or memory corruption based vulnerabilities allow an exploit to inject a set of instructions (shellcode) into the target process. Shellcode on its turn downloads an executable malware on the victim's hard drive and executes it. The executable installed on the target system is completely independent from the exploit pack (see [3] for some statistics on the pairings).

Figure 1 depicts the generic scenario of drive-by-download attack [3, 5]. A victim visits a compromised web site, from which he/she gets redirected to the exploit kit page. Various ways of redirection are possible: an <iframe> tag, a JavaScript based page redirect etc. The malicious web page then returns an HTML document, containing exploits, which are usually hidden in an obfuscated JavaScript code. If at least one exploit succeeds, then the victim gets infected. An exploitation is successful when the injected shellcode successfully downloads and execute a malicious program on the victim system.

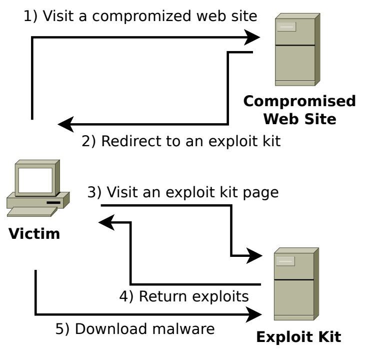

1) Visit a compromized web site
Compromised Web Site
2) Redirect to an exploit kit
Victim
3) Visit an exploit kit page
4) Return exploits
5) Download malware
Exploit Kit

Figure 1: Scheme of drive-by-download attack

3 Related Work

The infection dynamics enforced by exploit kits are presented by Provos et al. in [6] and Rajab et al. in [8]. An overview on the diffusion of exploit kits is also reported by Grier et al. in [3], where they analyse DNS traffic toward known malware-delivery domains, and by Allodi et al. in [1], where an analysis of attacks against vulnerabilities in the exploit kits is given. These works related on data recorded "in the wild" and differ therefore from ours in the experimental approach.

A technical analysis of Exploit kits and their capabilities are outlined by Kotov et al. in [5]. Details on heap spray attacks are given in [2], in which different exploits are analysed. However, no systematic experimental methodology is applied to *test* those exploits.

An overview of good practices when dealing with malware experiments is provided by Rossow et al. in [9]. Realism, safeness, data categorisation are all topics covered by the authors and considered in our work as well. On a similar, more active and investigation-oriented line, Kanich et al. [4] underline the difficulties of (safely) dealing with the cybercrime environment; however, while they are addressing multiple interactions with the underground, we are only measuring one snapshot of the underground community in our MalwareLab.

4 A (failed) first experiment design

Exploit kits are advertised on the black markets with a "self-declared" *infection efficacy*, expressed as the percentage of "incoming users" the costumer may expect to infect. The advertised ratios are usually around 15-25% [1]. The frequency and trends of successful exploitations depends on such factors as an operating system version, type and version of a browser and its add-ons, presence of security measures, and general system configuration.

From here we formulated our first question on exploit kits. *Question: How good are their exploits?* Our first experimental design was therefore to measure, by controlling the experiment for operating system and vulnerable applications, how successful each exploit was at infecting the vulnerable machine. We assumed that the state of the machine memory (in particular the browser's heap) played a role in the successfulness of the exploit. We manually set up about 40 vulnerable configurations per operating system (Windows XP and its Service Packs). In order to be sure to measure only *one exploit* at a time and avoid noise in the data, we paid extra attention to the pre-existing vulnerabilities on the system. For example CVE-2006-0003, a vulnerability widely exploited by most exploit kits, is a "system vulnerability" that exploit kits reach through the ActiveX interface of Internet Explorer 6. If we were testing for, say, a vulnerability in Java loaded *within the context of Internet Explorer 6*, then we would have had no control above a possible exploitation of CVE-2006-0003, which might have created the false impression that the Java vulnerability was successfully exploited. Avoiding such configurations required a significant amount of time. In the example above, we run the vulnerable Java plugin in a non-vulnerable (to our exploit kits) version of Firefox, and run multiple tests to be sure that Firefox did not play any role in the observed exploitation/non exploitation of the Java vulnerability.

We controlled the state of the browser's heap memory by automatically generating a random number of HTML + JavaScript web pages that were loaded onto the browser *before* sending a GET request to the exploit kit tested in that run. However, we ended up measuring only exploits that worked with 100% efficacy and exploits that simply *never* worked. When we studied this issue in more detail we found out that drive-by-download attacks exploit the fact that each browser tab is in fact a separate process (or a thread) with its own heap and memory disposition, so the browsing history does not impact the success of the attack.

Further investigations on the nature of exploit kits and the exploits they bundle revealed why the design failed. The exploits we tested either use Java vulnerabilities that use internal Java VM resources to download and execute malware (on which Windows defences have no effect) or use a heap spray attack. The heap spray attack is an exploitation technique against vulnerabilities in browsers and other applications that allocate user data in the process heap. In the case of browser attacks the idea of heap spray is to allocate hundreds of megabytes of payload in the heap memory (by creating a big number of JavaScript string variables) and then trigger the memory corruption vulnerability to redirect the process (that is the browser/tab/plugin) execution flow to the lower region heap. The injected shellcode consists of a huge NOP-sled (a set of NOP-like instructions) followed by the ac-

Table 1: Operating systems and respective release date.
*Configurations are right-censored with respect to the 6 years time window.

Op. system	Service Pack	Y_{sys}
Windows Xp	None	2001 - 2007
	1	2002 - 2008
	2	2004 - 2010
	3	2008 - 2013*
Windows Vista	None	2006 - 2012
	1	2008 - 2013*
	2	2008 - 2013*
Windows 7	None	2009 - 2013*
	1	2011 - 2013*

tual malicious code. Since the heap is now filled with the shellcode the probability that the instruction pointer will reach the address where the malware is loaded approaches 1. We conclude that exploits bundled in exploit kits are well engineered and designed to work disregard of the memory state of the victim machine.

5 Design of the Experiment

With the second experiment design we aim at giving a quantitative answer to the following research question:

Question: How resilient are exploit kits against software configuration updates?

To answer this, we *test* exploit kits in a controlled environment, our *MalwareLab*. The core of our design is the generation of "reasonable" *home-system* configurations to test against the infection mechanism and capabilities of exploit kits. We test those configurations as running on Windows XP, Windows Vista and Windows 7. Table 1 reports versions and release dates of each operating system and service pack considered (from here on, *system*). After an initial phase of application testing on the selected systems, we fix the life-time of an operating system to be 6 years for compatibility of software. Y_{sys} indicates the working interval of each operating system.

For our experiment we selected 10 exploit kits (see Table 2) out of the 34, leaked from the black markets, we gathered. Some of them proved to be not fully-functional or impossible to be deployed (e.g. because of missing functions). Out of those that were deployable and armed, we selected 10 according to the following criteria: (a) popularity of the exploit kit [10]; (b) year of release; (c) unique functionality (e.g. only one of multiple versions of the same kit family is selected).

5.1 Configuration selection

The automated installation of software configurations on each machine followed the definition of a criteria to select software to be installed. As often happens, this is

Table 2: List of tested exploit kits
For some exploit kits we could not find the respective release advertisement on the black markets, and therefore a precise date of release for the product cannot be assessed. For those (*) we approximate the release date to the earliest mention of that exploit kit in underground discussion forums and security reports. This identifies an upper bound of the release date.

#	Name	Version	Release Year
1	Crimepack	3.1.3	2010
2	Eleonore	1.4.4mod	2011
3	Bleeding Life	2	2010
4	Elfiesta	1.8	2008*
5	Shaman's Dream	2	2009*
6	Gpack	UNK	2008
7	Seo	UNK	2010
8	Mpack	0.86	2007*
9	Icepack	platinum	2007
10	Adpack	UNK	2007*

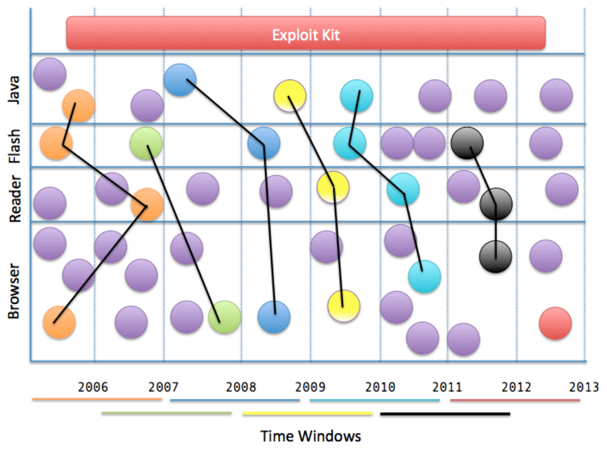
Table 3: Software versions included in the experiment.
Overall 9 software versions were excluded from the experiment setup because the corresponding installation package was either not working or we could not find it on the web.

Software	Versions	# of versions
Mozilla Firefox	1.5.0.2 - 17.0.1.0	122
Microsoft Internet Explorer	6-10	5
Adobe Flash	9.0.16.0-11.5.502.135	54
Adobe Reader	8.0.0-10.1.4	17
Java	1.5.0.7-7.10.0.0	49
	Total	247

subject to a number of assumptions that define the criteria themselves. For our experiment to be realistic, we need to build configurations that are *reasonable* to exist at a certain point in time. As an example, we consider unlikely to have Firefox 12, released in April 2012, installed on the same machine with Adobe Flash 9, released 6 years earlier in June 2006. We therefore fix a two-years window that defines which software can coexist. The window is based on the month and year of release of a particular software. Since our oldest exploit kit is from early 2007, we are testing software only released in the interval $(2005, 2013)$. Table 3 shows the software versions we consider[1].

Figure 2 exemplifies the mechanism to create configurations. The algorithm to generate each configuration iterates through all years Y_{conf} from 2006 to 2013, and chooses at random a version of each software (including "no version", meaning that that software is not installed for that configuration) that satisfy $Y_{swRel} \in [Y_{conf} - 1, Y_{conf}]$. For each Y_{conf} we generate 30 random configurations. Given the construction of Y_{swRel}, we end up with seven windows and therefore 210 con-

[1]We did not include Google Chrome as it was first released halfway through the timeline considered in our experiment (2008). Introducing Chrome samples in 2008 would have changed the probability of a particular software to be selected. In turn, this would make comparing time windows before and after 2008 statistically biased. We plan to include Chrome in future experiment designs.



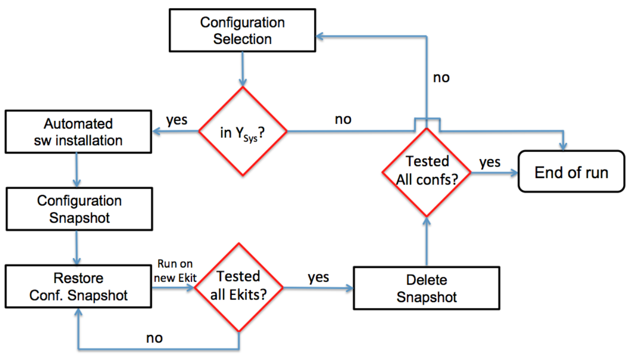
Each dot represents a tuple {*software, version*} that is eligible for selection to be included in a configuration in a certain time window. The probability distribution of each configuration is uniform (each tuple has the same likelihood of being selected). We exemplify the selection mechanism by highlighting the {*software, version*} tuples with the respective window colour. Note that because we treat "no software version" as a "version" in the tuple, software can as well be *not selected* for a certain configuration.

Figure 2: Random selection of configurations per software with sliding windows of two years.

figurations per system reported in Table 1. However, as underlined in Section 5, for compatibility reasons each system has a time window of 6 years starting one year before its release date. Because we want to measure the resiliency of exploit kits, we keep the number of configurations per year constant (otherwise results would not be comparable between different runs). This means that some systems are tested, overall, against a lower number of configurations than others. For example, Windows XP Service Pack 1 (2002-2008) will be tested only against configurations in the time windows{*[2006, 2008),[2007-2009))*}[2], which gives us 60 configurations. Windows Vista with no Service Pack (2006-2012) will instead be tested, for the same reason, with 180 configurations. This guarantees that each exploit kit is tested for each system against the same number of configurations per year.

The algorithm iterates through each configuration and runs it against the available exploit kits. Figure 3 is a representation of an experiment run for each system. At each iteration i, we select the configuration $conf_i$. If $Y_{conf_i} \in Y_{sys}$, we automatically install the selected software on the virtual machine using the "silent install" interface provided by the vendor or by the *msi* installer. A configuration install is successful when *all* software in that configuration is installed. For detection of unsuccessfully installed configurations, see Section 6.3.

[2]Note that the last year of the time window is not included. For example, [2006,2008) includes configurations from January 2006 to December 2007a.



This flowchart describes a full experiment run for each system in Table 1. Configurations are generated in chronological order, therefore if the first control on Y_{Sys} fails, every other successive configuration would as well and the experiment ends. Snapshots enable us to re-use an identical installation of a configuration multiple times.

Figure 3: Flowchart of an experimet run.

When the installation process ends, we take a "snapshot" of the virtual machine. Every run for $conf_i$ will restore and use this snapshot. The advantages of this are twofold: at first we eliminate possible confounding factors stemming from slightly different configurations, because only the exploit kit changes; secondly, this is also faster than re-installing the configuration every time, which would have considerably stretched the (already not short) completion time. When all exploit kits are tested, a new configuration is eligible for selection.

5.2 Data collection

In the course of our experiment we keep track of (a) the successfulness of the automated installation of a configuration on a victim machine (VICTIM) at any given time; (b) the successfulness of infection attempts from exploit kits. This data is stored in two separate tables, *Configurations* and *Infections* respectively.

1. *Configurations* is needed to control for VICTIM configurations that were *not* successfully installed; this way we can correctly attribute (un)successful exploitation to the right set-ups. This is desirable when looking for infection rates of single configurations or software.

2. *Infections* stores information on each particular configuration run against an exploit kit. We set our infection mechanism to make a call to the Malware Distribution Server (MDS) each time it is executed on the VICTIM machine. A "call back" to the MDS can in fact only happen if the "malware" is successfully executed on VICTIM. The MDS stores the record in *Infections*, alongside *(snapshot_id, toolkit_name, toolkit_version, machine, IP, date, successful)*. Exploit kits have an "ad-

ministrative panel" reporting infection rates [5]. However, we decide to implement our own mechanism because (a) it allows us to have more control on the data in case of errors or unforeseen circumstances; (b) exploit kits statistics may not be reliable (e.g. developers might be incentivated in exaggerating infection rates).

To minimise detection [3], some exploit kits avoid attacking the same machine twice (i.e. delivering the attack the same IP). This behaviour is enabled by an internal database controlled by the kit, independent from our *Infections* table. In some cases, e.g. when the experiment run needs to be resumed from a certain configuration, our *Infections* table may report un-successful attacks of an exploit kit, when instead the exploit kit did not deliberately deliver the attack in the first place. We therefore need to control for this possibility by resetting the exploit kit statistics when needed.

6 Operational realization

In this Section we present the technical implementation of our experiment design in its three key points: (1) virtualised system infrastructure; (2) automated execution; (3) operative data collection;

6.1 Virtualised System Infrastructure

When testing for malware, an isolated, virtualised infrastructure is desirable [9]. We set up a five machine network that includes a Malware Distribution Server (MDS) and four machines hosting the Victim Virtual Machines (VICTIMs). Initially, the setup also included an IDS and a network auditing infrastructure to log the traffic; however, to eliminate possible confounding factors caused by the network monitoring and auditing, we decided to eliminate this part of the infrastructure from the design reported here. For practical purposes (i.e. scripting), all machines are run on a linux-based operating systems, upon which the virtualised infrastructure is installed.

The purpose of the MDS is to deliver the attacks. Because of the nature of exploit kits, all we need to attack VICTIMs is an Apache Web-Server listening on HTTP port 80 upon which the kits are deployed. As mentioned, we implemented and armed the exploit kits with our own "malware", Casper.exe (our *Ghost-in-the-browser* [7]) to help us keep track of infected systems. In order to make it compatible with all Windows versions we have linked it statically with the appropiate libraries (e.g. *Winsock*). Casper reads a special configuration information file that we put on each victim machine and send its content to a PHP script on the MDS by using the *Winsock* API. This script (trojan.php) simply stores the received data along with the VICTIM IP address and timestamp into the *Infections* table in our database.

6.2 Automated execution

We use VirtualBox to virtualise victim machines. In order to automate the tests we take advantage of the tool that is shipped with VirtualBox called VBoxManage. It is a command line tool that provides all the necessary functions to start/stop virtual machines, create/delete snapshots and run commands in the guest operating system. The main program, responsible for running the experiment is a Python script that makes a sequence of calls to VBoxMange via subprocess Python module.[3]

At each run, our scripts read configurations.csv, a file containing all the generated configurations for that machine. The scripts iteratively install configurations upon the VICTIM system. The mapping between software version pointers in configurations.csv and the actual software to be installed is hard-coded in the core of the implementation. The automated installation happens via the silent install interface bundled in the installation packages distributed by most software vendors. However, because of a lack of a "standard" interface and the inconsistencies between different versions of the same software, we could not deploy one-solution for all software. We used instead a "trial-and-error" approach and online documentation to enumerate the arguments to pass to the installers and map them with the right software versions. Each configuration is then automatically and iteratively run against every exploit kit on the MDS.

Despite the experiment being completely automated, we found that some machines were failing at certain points in the run, most often while saving snapshots or uploading files to the VICTIMs. We therefore implemented a "resume functionality" that allows us to "save" the experiment at the latest valid configuration, and in case of failure restore the run from that point.

6.3 Operational Data Collection

To reset exploit kits statistics and guarantee the soundness of the statistics collected in the *Configuration* and *Infections* tables, we have implemented a PHP script that clears the records on delivered attacks the kit keeps. This step was rather easy to accomplish: we used the code snippets responsible for statistics reset in each exploit kit, and copy-pasted them into a single script.

We keep track of software installations on the VICTIM machines by means of a second dedicated script. To build it, we manually checked where each program puts its data on the file system at the installation. Because

[3]It should be noted that there is Python API for VirtualBox, that allows to run VirtualBox commands directly from within the Python environment. We tried to use it during our first (failed) experiment, but had to switch to VBoxManage, because Python VirtualBox API functions proved not to be very reliable on our machines.

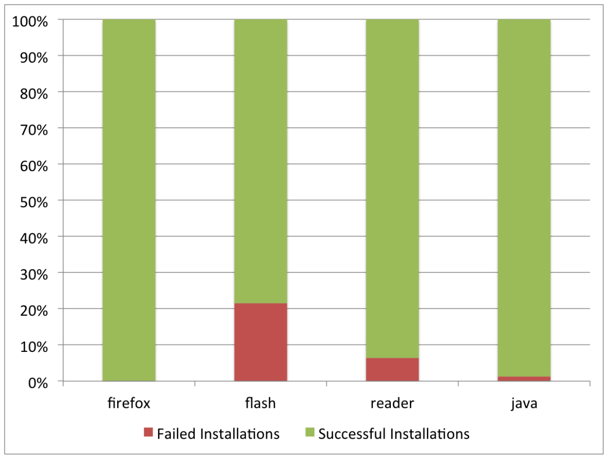


Figure 4: Stacked barplot of configuration installs by software.

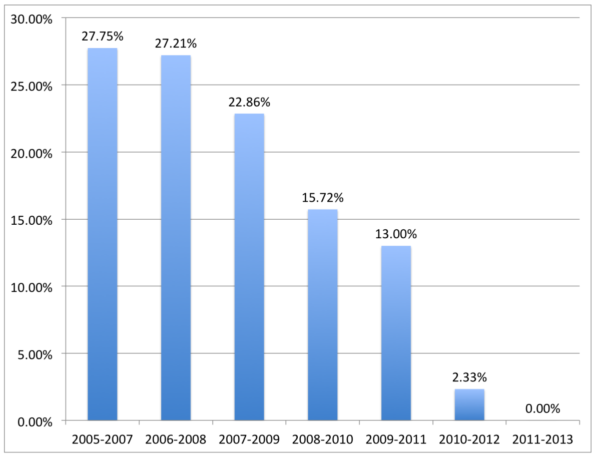


Figure 5: Infection rates per time window.

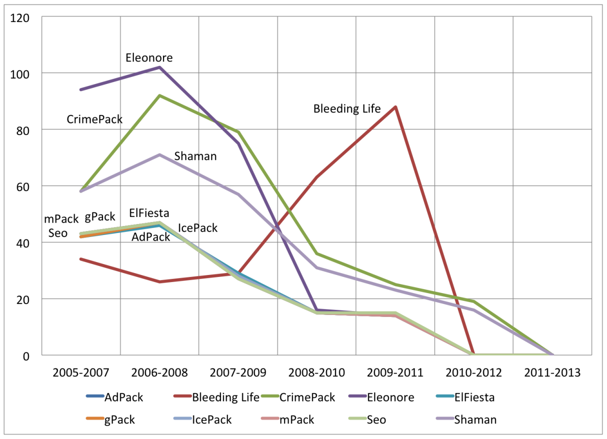


Figure 6: Infections per time window per exploit kit.

it was impossible to look at every application installation directory we sampled a subset of programs to check whether they always put data in the same place. Then we wrote a batch file that checks for the presence of the corresponding data directories *after* the alleged installation. The results of the batch file inspection are then passed to a Python script on the host machine, sent to the MDS, and stored in the *Configurations* table on our dataset.

To collect the infection data, when the MDS receives a call from a VICTIM machine, the MDS adds a record in the *Infections* table, setting the *successful* record to 0 (the default). When executed, *Casper* connects to the MDS via a PHP page we set up (namely infection.php). This updates the *successful* bit of the corresponding run record in *Infections* to 1.

7 Preliminary Experimental Results

The automatic installation procedure proved to be rather reliable. Figure 4 depicts a 100%-stacked barplot of configuration installs by software. As one can see, Firefox and Java were practically always successfully deployed on the machine. In contrast, 6% of Adobe Acrobat and 21% of Flash installations were reported to be not successfully completed. However, it proved practically unfeasible to manually check failures of our detection mechanism (e.g. the files for that software version on that configuration may be on a different location). We cannot therefore assess the level of false negatives our detection mechanism generates.

Figure 5 reports an overview of the infection rates of *all exploit kits* in each time window. Intuitively, because the exploit kits are always the same, the general rate of infection decreases with more up-to-date software. Observationally, from 2005 up to 2009 the success rate of exploit kits seem not to be affected by system evolution. A marked decrease in the performance of our exploit kits starts only after 2010. This observation is confirmed by looking at a break-up of volumes of infections per exploit kit per year, depicted in Figure 6. Generally speaking, each exploit kit (apart from Bleeding Life) seem to remain effective mainly within the first three time windows, from 2005 to 2009. Eleonore, CrimePack and Shaman lead the volume of infections in those years, with Eleonore peaking at more than 100 infections for 2006-2008, which amounts at about 50% of the configurations for that window. Interestingly, a few exploit kits seem identical in terms of performance. Seo, mPack, gPack, ElFiesta, AdPack, IcePack all perform identically throughout the experiment. Most exploit kits's efficacy drops in the fourth time-window, were configurations spanning from 2008 to 2010 are attacked. However, Bleeding Life is here an outliner, as its efficacy in infecting these machines rises and tops in 2009-2011 to more than twice its infection rates for 2005-2009. After 2011, however, its infection capabilities drop to zero. In the last but one time window (2010-2012), the only still effective exploit kits are Crimepack and Shaman. Overall three types of exploit kits seem to emerge:

1. *Lousy exploit kits.* Some exploit kits in the markets seem to be identical in terms of effectiveness in infecting machines. Not only they perform equally, but the identical trend throughout our experiment suggests that the exploits they bundle are themselves identical. This may indicate that some exploit kits may be rip-offs of others, or that an exploit kit author may re-brand the same product.

2. *Long-term exploit kits.* From our results, a subset of exploit kits (in our case Crimepack and Shaman) perform particularly well in terms of resiliency. Crimepack and Shaman are the only two exploit kits that remain active from 2005 to 2012, despite not being the most recent exploit kits we deployed (see Table 2). For example, in the period 2008-2012 Shaman performs up to two times better than Eleonore, despite being two years older. In other words, some exploit kits appear to be designed and armed to affect a wider variety of systems in time than the competition.

3. *Time-specific exploit kits.* As opposed to *long-term exploit kits*, some kits seem to be extremely effective in short periods of time only to "die" shortly after. Eleonore and Bleeding Life belong to this category. The former achieves the highest amount of infection per time window in 2006-2008, and drops then to the minimum within the next two years. The latter is the only exploit kit capable of infecting "recent" machines, i.e. those with configurations since 2009 on. Bleeding Life was in particular clearly designed to attack machines around the period of the release of the kit (2010).

8 Challenges and limitations

The experimental approach introduced with this work also introduces a number of technical and design challenges that we believe are worth further discussion.

1. Scale of the experiment. We collected only a limited amount of software to be tested in our configurations. This is due both to the lack of an automated way to collect "historical" software versions, and for deployment reasons: unfortunately, no standard interface to automatically deploy an application on a system exists. Using Microsoft MSI demonstrated to be not universally effective, as some (in particular old) software failed to be installed that way. We had therefore to manually check for the "silent install" arguments to give in input to the software install package. Apart from being a long operation to complete, it is also obviously quite prone to errors. These errors may be responsible for the failed installations reported in Figure 4. To enhance the scope of the experiment and configure more "realistic" machines with more software (e.g. Apple Quicktime, Real Player, Opera and Chrome) and respective versioning management, an automated way to gather and deploy such software would positively enhance both the representativeness and the reliability of the experiment. However, we are currently not aware of any efficient way to do that.

2. Configuration checking. Because serial, automated software installations may be subject to failure, a reliable detection mechanism for failed installs should be on top of the configuration setup process. Our implementation guarantees no false positives (i.e. configurations that were *not* installed but reported as such), but its quality in terms of false negatives is not clear. Still, to manually check its reliability is practically infeasible, given the volume of possible configurations to inspect.

3. Assumptions. This experiment is run on top of three main assumptions:
(a.) *Regardless of the release date of an exploit kit, the probability of attacking a configuration does not vary with the year of the configuration.* We believe that it is reasonable to assume that configurations from 2006 are still visible in the wild in 2012; however, to guarantee the realism of the experiment a "discount factor" should be applied to old machine runs. With the current design, older machines are run against old exploit kits with the same frequency as against new ones. This represents a threat to the realism of the experiment, as the probability of receiving a connection from a particular configuration is likely to decrease with time (i.e. $Pr_t(Conf_x) > Pr_{t+1}(Conf_x)$). This discount factor is however hard to assess, and further investigations may be needed to discard unrealistic or biased assumptions.
(b.) *Software configurations are valid solely within a certain time-frame.* We fixed a two-years time window for software deployment. We only have folk knowledge to support the validity of this assumption. This point is crucial as a different definition of "time window" may change the results of the experiment (e.g. see *long term* and *short term exploit kits* above).
(c.) *Exploit functionality, differently from malware's, is not affected by virtual machines.* We are aware that many malicious programs have virtual machine detection functionalities, that prevent them from being installed on virtual environments [9]. However, to the best of our knowledge no exploit detected in the wild and delivered by an exploit kit performs such actions.

4. Technical challenges. The major issue in implementing the malware experiment is making it realistic. In our design we make a victim browser visit the malicious web page and hold it for 60 seconds. In real world, however, a lot of exploits get delivered by pop up windows and banners which a victim user tends to close on sight [11]. Thus, introducing the 'time of exposure" of the victim to the malicious page could increase the realism of the experiment. Another aspect of a victim machine is its hardware configuration. We assign 1GB of RAM and 1-core processor for each VICTIM, but this may as well affect the realism of the experiment. For instance, we found that some exploits allocate so much data in the browser address space that on machines with <300MB of RAM the virtual memory and swap file of the VICTIM runs out, which eventually forces the kernel to kill the process. Assessing to what degree machine diversity is desirable is an open problem of our design.

9 Conclusion & lessons learned

In this paper we presented our MalwareLab, a platform to experiment with cybercrime attack tools. The presentation was focused on the technical and design issues we faced. As a result, we can synthesise the following *lessons learned* from our experience:

1. Exploit and possibly malware developers put extra-effort in enhancing the reliability of their products. This translates in hard-to-control experiments, because the number of possible confounding variables the experimenter should control for grows unpredictably.

2. Controls must be *checked* experimentally. Technically sound assumptions on the functionality of software artefacts may prove to be completely wrong or uneffective for the particular exploit or malware that is being tested.

3. Experiments always need a "resume point" from where restart failed runs. However, because of technical or operative requirements a resume point might not coincide with the last valid experiment run. To avoid biased measurements is important to assure that the experiment is restored to its original state at the "moment of the resume". This is particularly relevant when the experimental setup is distributed on a network of systems, and the experiment fails on a single, or subset of, machines.

At a technical level, we find that exploit kits show different capabilities in terms of resiliency of infections in time. Our answer to the research question outlined in Section 5 is therefore the following:

Answer: a clear distinction between high-quality and low quality products exists. Some exploit kits seem engineered to be effective for long periods of time, at the cost of lower top-rates of infection; other exploit kits instead seem developed to gather as many infections as possible in short periods of time, possibly around the time of their release. These kits achieve higher top-rates than the competition, but their resielincy is lower.

Acknowledgments

This work was partly supported by the projects EU-IST-NOE-NESSOS and EU-SEC-CP-SECONOMICS and MIUR-PRIN-TENACE Projects and the European Union by the Erasmus Mundus Action 2 Programme.

References

[1] ALLODI, L., WOOHYUN, S., AND MASSACCI, F. Quantitative assessment of risk reduction with cybercrime black market monitoring. In *In Proc. of IWCC'13* (2013).

[2] CORELAN, T. Exploit writing tutorial part 11 : Heap spraying demystified. `https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/`, checked on 27.04.2013.

[3] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., PROVOS, N., RAFIQUE, M. Z., RAJAB, M. A., ROSSOW, C., THOMAS, K., PAXSON, V., SAVAGE, S., AND VOELKER, G. M. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 821–832.

[4] KANICH, C., CHACHRA, N., MCCOY, D., GRIER, C., WANG, D. Y., MOTOYAMA, M., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. No plan survives contact: experience with cybercrime measurement. In *Proc. of CSET'11* (2011).

[5] KOTOV, V., AND MASSACCI, F. Anatomy of exploit kits. preliminary analysis of exploit kits as software artefacts. In *Proc. of ESSoS 2013* (2013).

[6] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. All your iframes point to us. In *Proc. of USENIX'08* (2008), pp. 1–15.

[7] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser analysis of web-based malware. In *Proc. of HOTBOTS'07* (2007), pp. 4–4.

[8] RAJAB, M., BALLARD, L., JAGPAL, N., MAVROMMATIS, P., NOJIRI, D., PROVOS, N., AND SCHMIDT, L. Trends in circumventing web-malware detection. Tech. rep., Google, 2011.

[9] ROSSOW, C., J. DIETRICH, C., GRIER, C., KREIBICH, C., PAXSON, V., POHLMANN, N., BOS, H., AND VAN STEEN, M. Prudent practices for designing malware experiments: Status quo and outlook. In *Proc. of the 33rd IEEE Symp. on Sec. & Privacy* (2012).

[10] SYMANTEC. *Analysis of Malicious Web Activity by Attack Toolkits*, online ed. Symantec, Available on the web at http://www.symantec.com/threatreport/topic.jsp?id=threat_activity_trends&aid=analysis_of_malicious_web_activity, 2011. Accessed on June 1012.

[11] WASH, R. Folk models of home computer security. In *Proc. of SOUPS'10* (2010), ACM.