ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
LABORATORY FOR COMMUNICATIONS AND APPLICATIONS 1
SEMESTER PROJECT

TRAFFIC ANALYSIS ON SOFTWARE PACKAGES

January 14, 2019

Student: Alexandre Dumur
Supervisors: Pr. Jean-Pierre Hubaux, PhD student Ludovic Barman

Contents

1 Introduction **1**

2 Background **2**

2.1 APT overview 2

2.2 APT packages metadata 2

2.3 Package Dependencies 2

2.4 Depends, Recommends and Suggests 3

2.5 Virtual package 4

3 Threat Model **5**

4 Attack overview **6**

4.1 Attack on HTTP request 6

4.2 Attack on Downloaded Size of packet traces 6

5 Theoretical results **8**

6 Attack methodology **11**

6.1 Technical details 11

6.2 Crawl 11

6.2.1 Extracting 12

6.2.2 Cleaning 12

6.2.3 Resolve dependencies 13

6.3 Capture 13

6.4 Attack 14

6.4.1 HTTP Matching 14

6.4.2 Size Matching 15

7 Practical Results **17**
7.1 Attack on HTTP Request (Strong Assumption) 17
7.2 Attack on Size (Weak Assumption) 17

8 Conclusion **22**

Bibliography **24**

Abstract

In this report, we explore the feasibility of inferring package downloads via traffic-analysis. Most downloads happen over HTTP, making the task trivial for an attacker. We show that even on top of HTTPS, sizes leak enough information to identify packages.

1. Introduction

Package managers are used because they provide a handy interface that allows the user to download, remove or upgrade software packages easily. Most of them provide some mechanisms that address security concerns such as authenticity, integrity and privacy. Nevertheless, some researches have already been conducted regarding security issues in package managers. J. Cappos et al. showed that the lack of protection in some package managers allows an attacker to push the victim to install a crafted package, or retain the victim from upgrading packages [1]. However, most attacks consider an active attacker. In this work, we consider a passive adversary which observes the (possibly encrypted) traffic from a target machine. Examples of this adversary include coworkers, system administrators, ISPs, etc. In the area of software packages, privacy is a serious matter. First, because the type of software packages installed on a particular system can reveal sensitive information about the profile of the one who owns the system. But more importantly, an attacker who knows which software package is installed in a particular system, can exploit some known vulnerabilities about that software package.

Throughout this report, we assume that the victim is installing software packages using Advanced Package Tool (APT). There are several reasons why we chose APT: first, it is widely used, as the default tool in Debian-based systems [2], and it has been shown to be relatively secure in comparison to alternatives [1].

This report is organized as follows: in Section 2, we give the required background knowledge about APT. In Section 3, we explain our Threat Model. In Section 4, we give an overview of the attack. In Section 5, we show some results that satisfy necessary conditions for the attacks to work. In Section 6, we explain what are the steps that an attacker needs to follow in order to guess what a victim is installing. In Section 7, we show some practical results that we collected after simulation of the attack.

2. Background

In this section, we provide essential knowledge about APT so that the reader has a complete understanding of the critical points that an attacker has to consider to carry out his attack.

2.1 APT OVERVIEW

As stated in the introduction, APT comes by default in Debian-based systems. It provides a graphical front-end (command line interface) that allows the user to remove, upgrade, download and install packages. Thus, APT is responsible to fetch packages that the user requires from remote repositories through HTTP or HTTPS depending on the settings. Once the package is downloaded, APT invokes *dpkg* which manages packages that are available locally.

2.2 APT PACKAGES METADATA

The information about packages' metadata are available in `/var/lib/apt/lists/`. This directory is kept up-to-date, with the command `apt-get update`. This will fetch the fresh packages' metadata by contacting the Debian package repositories listed in the directory `etc/apt/souces.list.d` and in the file `ect/apt/sources.list`.

Thus, the directory `var/lib/apt/lists` contains, for each package, multiple information. For example, we have the following fields: Package Name, Package Version, Size, Installed Size, Maintainer.

2.3 PACKAGE DEPENDENCIES

One popular asset of package managers is that they resolve package dependencies. Indeed, a particular software might require other software packages in order to run or to be fully

exploitable. Consequently, when a user asks APT to download a particular software package, APT first looks for its dependencies (in `var/lib/apt/lists`). If all the dependencies of that package are already installed, APT will only download and install the package that the user initially wanted to install. If some dependencies of that package are missing, APT will also download and install the missing packages.

In APT, dependency fields are represented in the following format: *package_name (* package_version)*. Where *package_name* is the name of the package, and * is the relation about the specific package version *package_version* [3]. The relations allowed are the following: <<, <=, =, >=, >>. If *(* package_version)* is not specified, any version can by installed. When multiple dependencies are required, they are simply separated by a comma. Sometimes, one package can be installed instead of another. In this case, the two packages are separated by a vertical bar: "|". We illustrate what has been said by the example bellow:

```
Package:  mutt
Version:  1.3.17-1
Depends:  libc6 (>= 2.2.1), default-mta | mail-transport-agent
```

We see that the package mutt Version 1.3.17-1 depends on any package `libc6` with version greater or equal to `2.2.1`, and needs either `default-mta` or `mail-transport-agent` with no requirements on the version [3].

2.4 DEPENDS, RECOMMENDS AND SUGGESTS

We identify three main classification of dependencies as it is important for our attacker to be aware of.

- ***Depends***. For a particular package A, the *Depends* packages of A are the packages that have to be installed for A to run.
- ***Recommends***. For a particular package A, the *Recommends* packages of A are the packages that without them most users will not use package A.
- ***Suggests***. For a particular package A, the *Suggests* packages of A are the packages with content related to A.

By default, APT installs the *Depends* and the *Recommends* packages but not the *Suggests* packages. Therefore, we consider the dependencies of a package as being the *Depends* and *Recommends* packages. We note that the flags `-install-suggests` allows the user to manually install suggested dependencies and `-no-install-recommends` to not install the recommended dependencies. However, it seems safe to assume that most user do not use this variant.

The fields *Depends*, *Recommends* and *Suggests* are available in the package's metadata in `/var/lib/apt/lists`. The problem is that those fields only take into account the first-level dependencies of a package i.e. it does not take into account dependencies of dependencies. For example, if package x depends on y and z, and y depends on w, then the *Depends* field of x will only be y, z and not y, z, w (in other words, only first-level dependencies are indicated in a package metadata). This forces the attacker to perform extra work to discover dependencies.

2.5 VIRTUAL PACKAGE

Sometimes, a package needs a specific functionality that many packages possess. In this case, since a lot of packages can provide this functionality, APT uses the abstract notion of virtual package. A virtual package is then a generic package name (it does not physically exist) which applies to a group of package that provides a same functionality. For instance, www-browser is the virtual package that provides a web browser, and thus `konqueror` and `firefox-esr`, which are web browser programs, apply to the same virtual package `web browser` [5]. Therefore, if any package have the dependency `www-browser` APT can resolve this dependency by installing either `konqueror` or `firefox-esr`.

3. Threat Model

In this section, we discuss the assumptions that are made about the attacker and the victim. We present two scenarios: One with **weak assumption**, and one with **strong assumption**. As we explained in the introduction, we assume a victim running a Debian-like system and using APT. We consider an adversary who is passive. Finally, we also assume that:

- ***The attacker has access to the victim's traffic.*** There are many ways for the attacker to get access to the traffic. For example, the attacker can be on the same WLAN, or might have access to a router.
- ***The attacker can distinguish exactly when a packet trace starts and ends***. This assumption ensure that the attacker does not mix packets coming from different packages installation. There are also solutions that have been derived to that sort of problem. Tao Wang and Ian Goldberg suggest a splitting algorithm that can split two packet traces using inter-packet time delay and machine learning techniques [7].
- ***The Attacker is able to filter out the packets that are not related to the victim's package installation*** . This could be done for example by creating a set of IP addresses from Debian package repositories and filtering out the packets with destinations that do not belong to the set.

With **strong assumption**:

- ***The attacker has access to the packets' content.*** This is the case when the victim is not using HTTPS and any other encrypted schemed protocol, or when the attacker can decrypt the content by himself.

With **weak assumption**, the attacker does not have access to the HTTP contents of packets. Obviously, the **strong assumption** will provide better results since the attacker will use the HTTP content to derive a better attack.

4. Attack overview

From the previous section, we have two kinds of scenarios: with strong assumption and weak assumption leading to two kinds of attacks: ***attack on HTTP request*** and ***attack on downloaded size of packet trace***.

4.1 Attack on HTTP request

The attack on HTTP request assumes that the attacker has access to the HTTP content of the packets captured on the network. The attack goes then as follows: The attacker will extract all the HTTP GET request from the packets that are captured during a package download. Note that the attacker might extract an arbitrary number of GET Request emanating from the execution of `apt-get install p`, where `p` is the package that the victim wants to install, since `p` may have dependencies that the victim needs to install before installing `p` (Section 2.3). Once the attacker collected the HTTP GET requests, he simply finds the match of the request url against one of the field of package's metadata available in his database.

Fig. 4.1 illustrates this attack.

4.2 Attack on Downloaded Size of packet traces

With weak assumption, the attacker no longer has access to the HTTP content of the packet. The technique we derived is based on the cumulative sum of the size of TLS/TCP payload that the server sends to the client. We will see in section 5 that this sum which represents the size of the package and its dependencies has a high level of uniqueness. We note that we can also use size matching with the attack on HTTP request (previous subsection). Moreover, we will find the exact size of a downloaded package by only looking at the HTTP 200 OK response in the field content-length.

Fig. 4.2 illustrates the attack on downloaded size of packet traces.

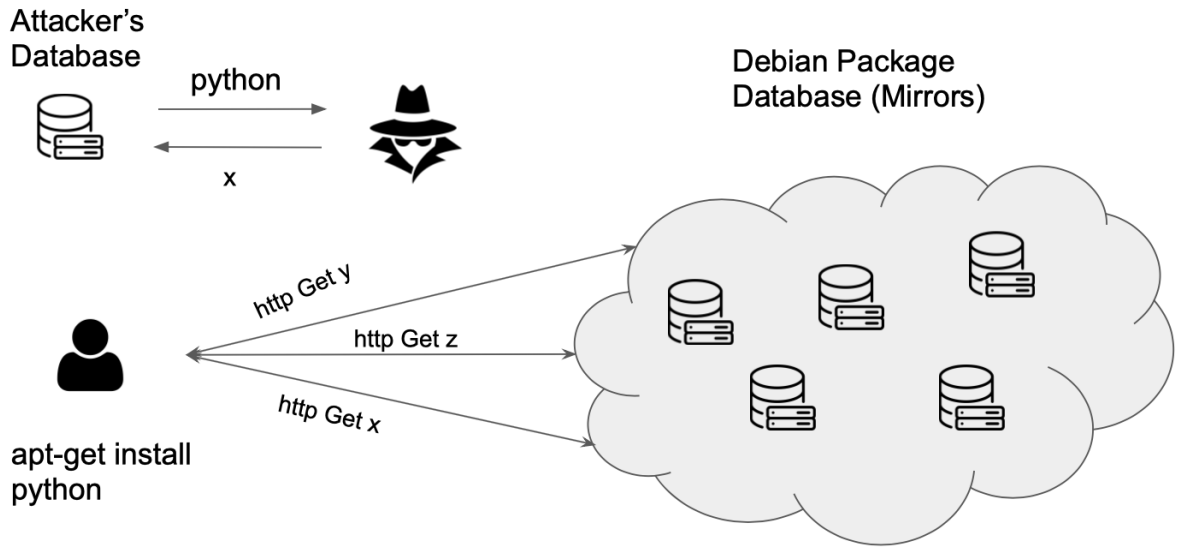

Figure 4.1: Attack on HTTP GET request. Attacker (black hat) collect every http request and look if the GET field matches any field in his database. In this example, the victim uses `apt-get install python` which generates 2 dependencies. The field x in the HTTP GET request is the correct match in the attacker's database.

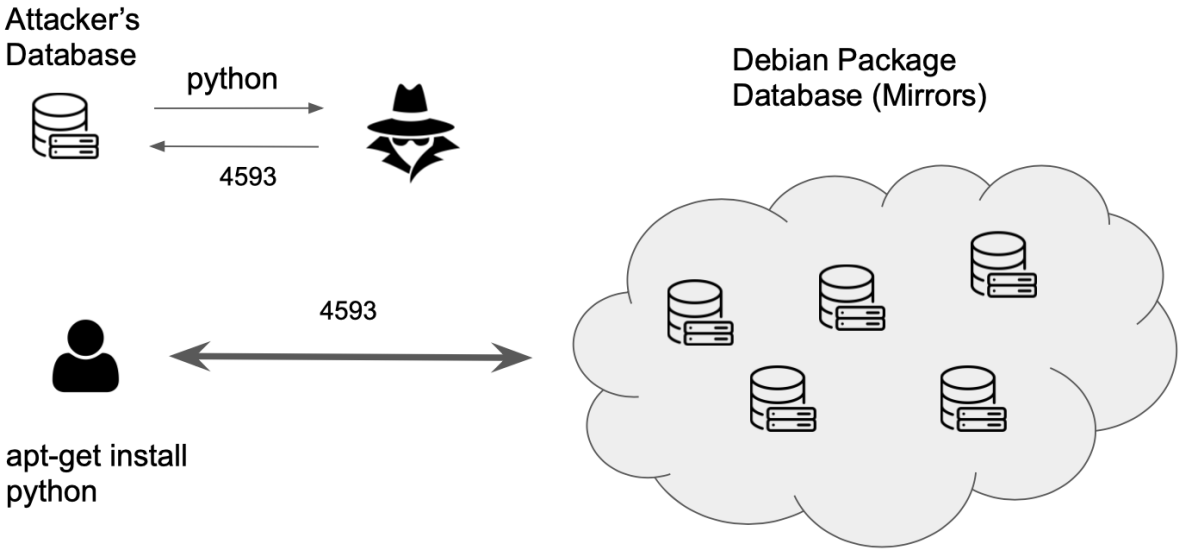

Figure 4.2: Attack on Downloaded Size of packet traces. Attacker (black hat) sees the total downloaded size once the download is completed, but cannot distinguish between different flows. The attacker then matches the total captured downloaded size to his database

5. Theoretical results

Here we present the necessary conditions for the attacks in the above section to be feasible.

Attack on HTTP request
For the first attack presented (Section 4.1), the necessary condition is that there exists a field in the attacker's database that matches the HTTP GET field. This field is called Filename and it is actually what ATP uses to craft the HTTP GET request. This condition is satisfied.

Attack on Downloaded Size of packet trace
The necessary condition for the second attack is that the size of packages are unique and thus can identify reliably a package. The more size collision the less accurate will be the attacker. Our task is now to collect the size of all the packages to see if the number of size collisions is acceptable for an attack. Fig. 5.1 shows the CDF of the percentage of packages according to their size uniqueness level. We see, at the first glance that already more than 94% of the packages are unique and then 97.5% are unique or tuples. Clearly this is acceptable for an attacker.

However, as we know from 2.3, one package installation launch with APT can trigger others. It is therefore a judicious choice to not stop the theoretical section yet and ask ourselves if size uniqueness is not hurt when we also consider the dependencies of packages in the size. Fig. 5.2 shows the same as Fig. 5.1 except that this time we are considering all the dependencies in the size.

Here again, the result is straight forward to analyze: the attacker does not have to worry when all the dependencies are considered in the size. However, there is still a problem. We only considered two extremes : either the victim has all the dependencies installed, in such case Fig. 5.1 is the right plot to analyze or the victim does not have any dependencies installed and it is Fig. 5.2 that we need to look at. In practice, the state of the victim might be in between

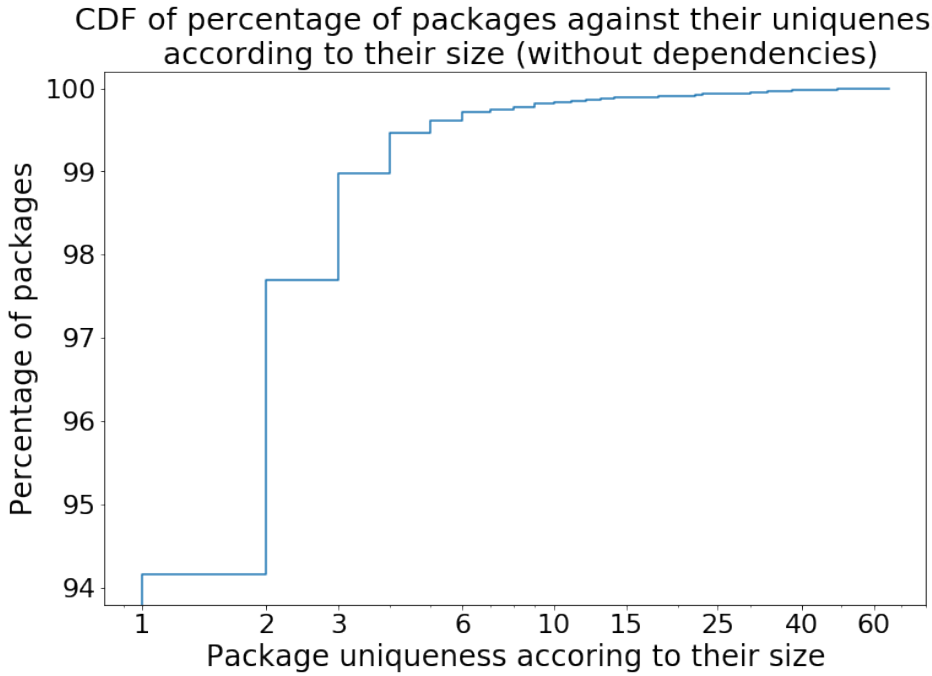

Figure 5.1

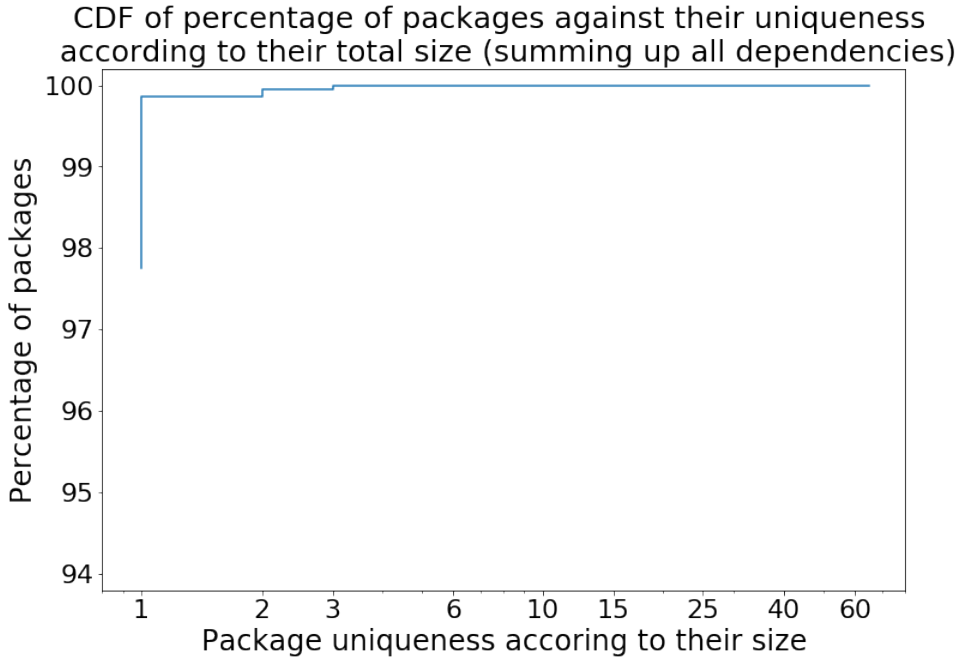

Figure 5.2

those two cases. The victim can have some dependencies installed and some dependencies that he still needs to resolve. The natural way of thinking about a state which is in between

the extremes is to look at the package's frequency of being part of the dependencies of other packages. If a package is present in a large number of other package's dependencies, it will be more likely that the victim will eventually install it. Therefore, in Fig. 5.3, we consider in the size of packages only the dependencies that have a smaller frequency (as being part of other package's dependency) than a fixed frequency `freq` for 3 different values of `freq`.

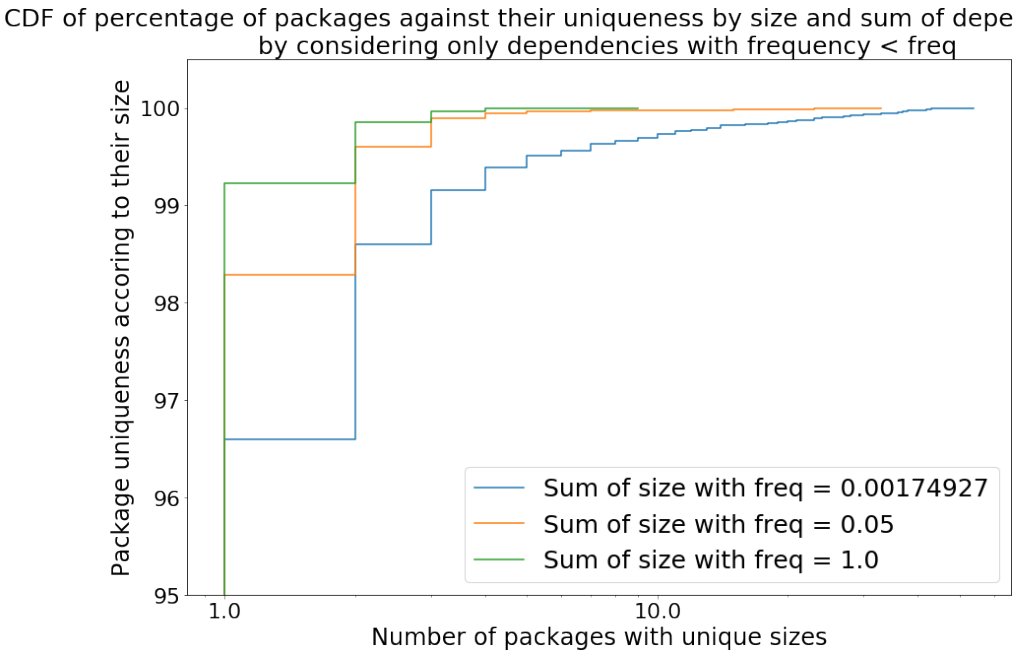

Figure 5.3

Looking at each curve of Fig. 5.3 confirms us that the lower the frequency `freq` is, the lower the number of dependencies taking into account, and therefore, the closer we are from Fig. 5.1. The inverse holds as well: if the frequency `freq` is higher, we are considering more dependencies in the size, and therefore the plot is closer from Fig. 5.2. This is also a good result for an attacker. The plots generated by the three victim states (with freq = $\{1.0\%, 0.05\%, 0.00175\%\}$) still stay in the two extremes, and thus provide a good level of uniqueness.

6. Attack methodology

We have seen that the two attacks are theoretically achievable, so we provide, in this section, the procedure to perform them. For that, we will first give technical details about how such attacks can be simulated. Then, we explain the attack procedure in 3 separated sections:

1. **Crawl**: Build the attacker's database used for matching a capture of packet trace to a software package (name and version).
2. **Capture**: Capture, on the network, the packet trace that corresponds to a package installation.
3. **Attack**: Find a match between the captured packet trace and a software package in the attacker's database.

6.1 TECHNICAL DETAILS

In our simulation, the victim is represented using a docker container, and the attacker is the host machine. The victim downloads packages using `apt-get install` with options `-d` to ensure that packages are only downloaded and not installed. Each time before launching a download with `apt-get install` on the victim's container , we start recording every packet that goes in or comes from the victim's container using `NetfilterQueue`. Once the download is finished, we run `apt-get clean` to clean the cache for further download. This will ensure to re-download the package if it belongs to dependencies of other packages in future captures.

6.2 CRAWL

In this section, we show how to build the attacker's database. We can divide the Crawl section into 3 subsections:

1. **Extracting.** Consists of extracting the raw package's metadata, and filling the attacker's database with the raw packages' metadata.

2. **Cleaning.** Consists of cleaning the database extracted in the previous step.

3. **Resolve dependencies.** Find all dependencies of a package (not just the first-level dependencies, Section 2.3)

6.2.1 Extracting

The very first step that the attacker has to do is to extract the metadata of packages available in `var/lib/apt/lists`. But before that, he must make sure that this directory is up-to-date by running `apt-get update` (Section 2.2).

An important action that the attacker must not forget is to mark the initially-installed packages in his database. This is the basic state of a victim and will be taken into account for the Size Matching in section 6.4.2. In order to collect the initially-installed packages, an attacker can create a new instance of Debian system and run `apt list -installed` to see which packages are already installed.

After having crawled the metadata, the attacker is supposed to have in his database the whole crawled packages with full features as well as the a unique identifier package_id which will be used to match a unique package to a capture.

6.2.2 Cleaning

In 6.2.1, the attacker ends up with a raw database of the crawled packages' metadata. Many attributes of package's metadata are useless for the attacker. He might want to get rid of them. (i.e. the attacker may not want to keep the 'Maintainer' field). An important field to keep is the field Filename (Section 5) since it will be used by APT to generate a HTTP GET request. It is therefore obvious that this field will be useful for an attacker who has access to the full content of packets.

While doing a capture, the attacker will most likely only be interested in knowing the package's name and version that the victim is installing. Therefore, it makes sense to keep only one unique package's metadata record per tuple: (size, name and version). This operation will reduce the confusion of having two or more packages with the same size, package name and package version. This will also reduce the attacker's database and hence, the time to resolve the dependencies and the do the size matching (Section 6.2.3 and 6.4.2 respectively).

6.2.3 Resolve dependencies

As we know from 2.4, the *Depends* and *Recommends* field extracted from `/var/lib/apt/lists/` only consider the first-level dependencies of packages. By only considering the first-level dependencies packages, the attacker will build a bias version of his database with less dependencies than what actually packages contain. Therefore, we designed the algorithm Alg. 1 which captures the full dependencies of a package by visiting all the dependencies of that package recursively.

Algorithm 1 Find all dependencies of a package given its first level dependencies

Input: s_{in}, set of first level dependencies of a package
Output: s_{out}, set of all dependencies of the underlying package

```
1:  function RESOLVEDEPENDENCIES(s_in)
2:      s_out ← s_in
3:      for d in s_in do
4:          s_d ← GETFIRSTLEVELDEPENDENCIES(d)
5:          if LEN(s_d) == 0 then return s_out
6:          end if
7:          s_r ←RESOLVEDEPENDENCIES(s_d)
8:          s_out ← s_out ∪ s_r
9:      end for
10:     return s_out
11: end function
```

Where the function `getFirstLevelDependencies(`d`)` takes a package as input and return the list of his first-level dependencies by parsing the *Dependens* and *Recommends* field (Section 2.4).

We note that the above algorithm is a simplified version of the one we used. In practice, APT dependencies do not form a tree and contain loops. For instance, a package x can have y as dependency and y can have x as dependency. So in the original algorithm, we make sure to stop the recursion when one package has already been seen.

6.3 CAPTURE

Given our threat model, the attacker, when doing a capture of packet trace, is only interested in either the GET field of the HTTP request, or the total downloaded size.

The HTTP GET field is trivial to capture. But an attacker has to be careful when downloading the TCP/TLS payload for an attack relying on the size of the packet traces. Indeed, an attacker can see duplicate packets that he has to drop. One other thing that the attacker has to take care of is that by taking the whole HTTP payload, the attacker includes the HTTP response header which is not part of the size of a package (in our experiment, this header size is around 280±7). This is an important fact, since the captured packet trace will be a bit bigger than the actual package size. Also, we note that the uncertainty will grow as the number of dependencies that need to be installed grows. Since each package dependencies will need HTTP response header and therefore HTTP header will come along. Of course, the attacker has to make sure to only count the downstream payload in the size of the capture packet trace.

6.4 ATTACK

In this section, we assume that the attacker has in his database for each package the list of the dependencies resolved (Section 6.2.3) as well as the Filename field. He also collected on the network the size of packet traces he wants to guess (Section. 6.3).

6.4.1 HTTP Matching

For HTTP matching, the attacker just needs to make sure to format the HTTP GET request to the Filename. Bellow is an example of the HTTP GET request and its associated Filename field in the database:

- GET /ubuntu/pool/main/d/dee/libdee-doc_1.2.7%2b14.04.20140324-0ubuntu1_all.deb HTTP/1.1
- pool/main/d/dee/libdee-doc_1.2.7+14.04.20140324-0ubuntu1_all.deb

From the two lines above, we note that the attacker has to get ride of GET /ubuntu/ and HTTP/1.1. We notice also that "%2b" corresponds to "+". Therefore, the attacker has to HTTP decode the GET field before processing to the match. A complete list of http encoding/decoding characters are available in [6]

6.4.2 Size Matching

The approach we will use to match the size of a packet trace to a software package will be to compute the distance (Eq. 6.1) between the size of the packet trace captured on the network and the expected size that would have the packet trace if the client ask APT to install it. The lower the distance is, the more confident the attacker is about the package to be the correct guess.

This distance function has to take into consideration the state of the victim. Indeed, the state influences the number of packages to be installed. Depending on the already installed packages APT will download more or less dependencies (less if the victim has a lot of packages already installed more, if he has just a few (Section 2.4)).

As mentioned in 6.2.1, the initial state can be crawled or estimated (by the frequency of packages as belonging in other packages dependencies). After that, each time the victim install a package, the attacker can update the state of his database by marking the installed package and dependencies that come along (with a boolean bit for instance). In this way, the attacker remember which packages to not take into account for further captures.

We present, in equation 6.1 the distance formula that is applied to each package j in order to find the aforementioned distance.

$$\text{dist}(j) = |s_{captured} - \delta(|\mathscr{D}_\text{j} \setminus \mathscr{M}| + 1) - s_j - \sum_{i \in \mathscr{D}_\text{j} \setminus \mathscr{M}} s_i| \tag{6.1}$$

Where:

- $s_{captured}$ is the captured size of packet traces
- $\mathscr{D}_\text{j}$ is the set of dependencies of package j
- s_i is the size of package i
- δ is the average size of a http header.
- $\mathscr{M}$ is the set of marked packages (packages that are already installed by the victim, i.e. state of the victim)
- $\mathscr{D}_\text{j} \setminus \mathscr{M}$ is the set of dependencies of package j without the marked packages.

Into words, Eq. 6.1 is the absolute difference between the captured size of the packet trace ($s_{captured}$) minus the expected size of packages that need to be downloaded ($s_j + \sum_{i \in \mathscr{D}_j \setminus \mathscr{M}} s_i$) including the size of the extra headers ($\delta(|\mathscr{D}_j \setminus \mathscr{M}| + 1)$).

Once the distance function is applied to every package, the attacker can sort in an ascending order all the packages according to the distance value. His first guess will be the first software package.

We note that once a package has been detected by the attacker either from the attack on HTTP request, or by the attack on size, the attacker will also know that all the dependencies of the detected package are installed on the victim's system.

7. Practical Results

In this section, we present the results of the two simulated attacks from Sect. 6. In order to do so, we divide the results in 5 categories, where in each category the attacker will capture 100 packet traces of software package that has {0, 1, 6, 10 and 15} dependencies. After that, the attacker will launch the two attacks.

7.1 Attack on HTTP Request (Strong Assumption)

The table Tab. 7.1 shows the results of the attack on the 5*100 captures separated by dependencies.

We see that the attack has a very high accuracy: 95.8% of the 500 guessed packages are correct. There is still a little amount of it that fails. This could be explained by a mismatch of the HTTP GET field to the Filename field due to HTTP formatting (Section 6.4.1). We note that the attack does not seem to be influenced by the number of dependencies of packages.

# Dependencies	0	1	6	10	15
Succeeded	98	94	94	98	95
Failed	2	6	6	2	5

Table 7.1: Results of the http matching Attack for the experiment described previously.

7.2 Attack on Size (Weak Assumption)

Fig. 7.1 shows for each of the 5*100 captures, the CDF of the percentage of packages against the rank of the correct guess. To have a better understanding of that graph, we give the

following example: when we look at the green curve at position 4 in the x axis, we can read approximately 70%. This indicates that for packages with 6 dependencies, 70% of the time, the actual package that the victim installed lies in the top 4 guessed packages. In short, the bigger the area under the curves is, the more accurate is the attacker in his guess.

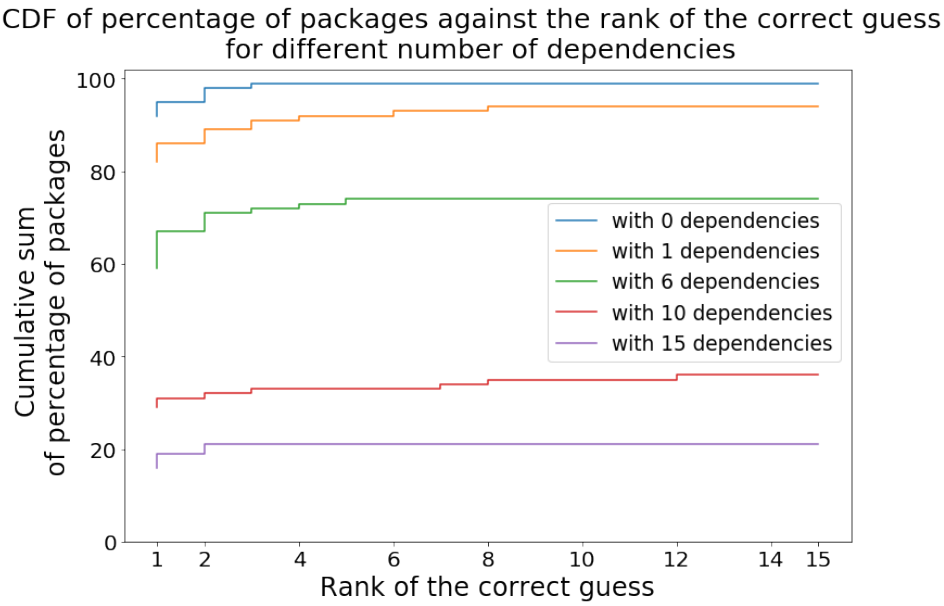

Figure 7.1: Results of the simulation described in the beginning of this section. Each curve is drawn from 100 captures and associates rank of the correct guess.

The first analysis we can make on Fig. 7.1 is that when we do not have any dependency in the package the victim installs (blue curve), our practical results match fairly well the theoretical result in 5. Indeed, more than 95% of the packages installed with 0 dependencies are the first guess of the attacker.

However, the more dependencies the less accurate the attacker will be in his guessing. Indeed, only 20% of the installed packages with 15 dependencies have as their first ranked guess the correct guess. This result comes in opposition to our theoretical results where we showed from Fig. 5.2, that the size is more unique when there are more dependencies to consider. This results from various facts that we will explain later in this section. We see that in our graph, the attacker is either correct, or totally wrong (as the tendency of the curves is flat).

To validate this hypothesis, we consider the distance value (Eq. 6.1) used to rank the attacker's guess on packages. For that, we will look at some statistics on Tab. 7.2 made on two categories of results from our 500 captures: the one with correct guess ranked <= 15 and

the one with correct guess ranked > 15. We can see that indeed, when the correct guess is ranked < 15, the distance value of the first ranked guess is by far lower than for > 15. The mean is approximately 8 times lower. If we look at the median and the quartiles, here again, the distance is significantly lower. We can also see how this distance is not stable for packages with bad ranked correct guess: the standard deviation is about 8 times higher than the good ranked correct guess.

	correct guesses ranked <= 15	correct guesses ranked > 15
mean	27.2	656.7
std	267.4	2104.5
min	0	0
25%	1	8
50%	2	45
75%	3	308
max	3870	20548

Table 7.2: Analysis on the distance of the first guessed package for correct guess ranked < 15 and correct guess ranked > 15

This indicates that the adversary is also able to understand if his attack worked or not. For instance, if the attacker sees that the distance of the first guess is 300, the chance that his attack failed will be significantly higher than if the attacker sees 1 as the distance of the first guess.

The explanation about why the packages with more dependencies are the ones with lower accuracy, as opposed to the theoretical result in 5, becomes clear. It comes from the fact that the attacker does not always succeed in resolving all the same dependencies that APT resolves, and thus is left with a lower number of dependencies than the true value, or the same number but with different instances of what APT resolved. Consequently, our distance formula is not robust to such errors, as it might miss whole packages, or consider wrong size of packages. Obviously, the bigger the number of dependencies a package contains, the bigger is the number of dependencies that the attacker could wrongly include or exclude.

We identify two reasons why the attacker can mismatch packages and they both come from the function `getFirstLevelDependencies(`d`)` in 6.2.3. As a reminder, the task of this

function is to: take a package, go to its *Depends* and *Recommends* field, translate the text into real package instances and return them. As we know from 2.4, the field representing one dependency specification is not an injective mapping. Indeed, only the case where the relation of the package version is "=" leads to a unique package. Nevertheless, in most cases, the relation is different from "=" and thus letting the unction `getFirstLevelDependencies`(d) many possible choice for returning package instances. This can provoke mismatch between what APT will ask to download and what the attacker guess that APT will download. The proportion of relations overall dependencies is shown in Fig. 7.2. The other reason is similare to the first one but is due to virtual packages. As mentioned in 2.5, virtual packages can be resolved using many different packages that have a common functionality. Here again, the attacker is not sure about which one APT will choose.

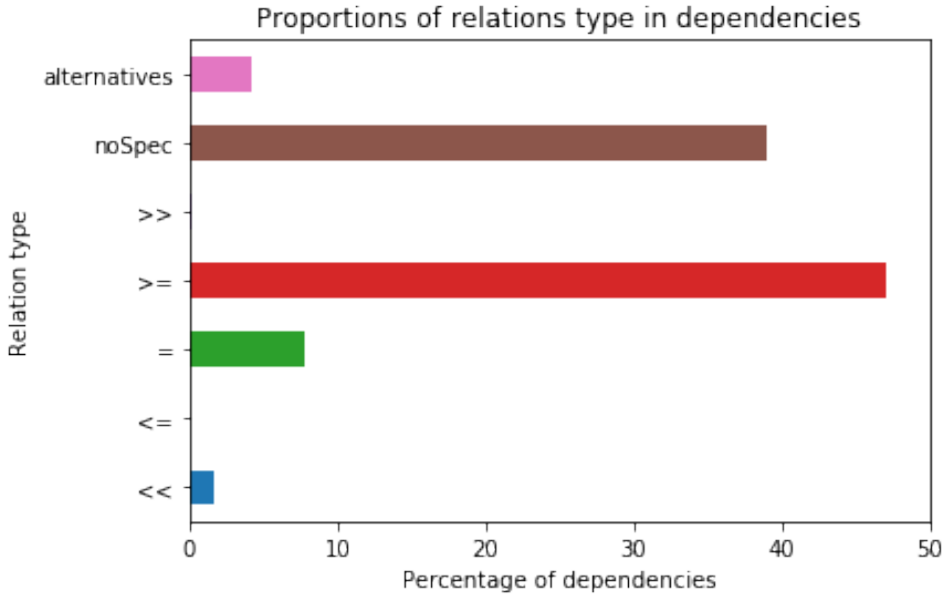

Figure 7.2: Proportion of the relation type for package's version in all dependencies. Alternatives corresponds to the vertical bar "|" (Section 2.4)

To conclude this section, we note that in Fig. 7.1, the variation in each curve is due to the uncertainty coming from the extra HTTP header size (δ in Eq. 6.1), whereas the drop of accuracy between the curves, comes from the difficulty for the attacker to resolve first-level dependencies. We also make a brief comment on the max of the first column of Tab. 7.2. 3870 for the distance size of the first guess seems to be too large for the correct guess to be <= 15. However, the following scenario can still happen: the attacker indeed can miss a dependency but even with that, the size of the dependency that attacker missed is still

small enough such that there is not more than 14 other packages with distance lower than the correct guess package. For completeness, we include in the appendix the cumulative distribution of packages by number of dependencies.

8. Conclusion

In this report, we saw two kinds of attacks targeting a machine that is performing software installation through APT. We assumed that the attacker has access to the traffic, can filter the packets emanating from the software installation and is passive. We resume the pros and cons of both attacks. We then discuss the possibility of improving the attacks.

ATTACKS: PROS AND CONS

Attack on HTTP GET request
This attack is weaker than the second one since it assumes that the attacker has access to the HTTP GET content. However, by default APT does not use HTTPS, and it is well known that most of the users do not change the security settings. Moreover, most of the Debian repositories do not even support HTTPS. We could therefore expect that this attack is applicable in most cases. Moreover, this attack shows very good results (Tab.7.1), does not require a lot of effort from the attacker, and is of low complexity for crawling and for matching, as it requires only to go through the database once.

Attack on Downloaded Size of packet traces.
This attack is stronger than the first one in the sense that it does not assume that the attacker has access to the HTTP content. Naturally, the accuracy of this attack is strictly lower than the previous one (Fig. 7.1). But we can still achieve a good level of accuracy if the package to guess does not have a lot of dependencies to resolve. Nevertheless, this attack requires a good estimation of the state of the victim: which packages are already installed in the system. In the specific scenario, the task is trivial, but the general problem is non-trivial. We emphasize that the adversary can recognize when his assumptions about the machine state are wrong (due to the discrepancy in the distance showed in section 7.2)

IMPROVEMENT OF THE ATTACK

Both of the attacks can be improved. For the first one, because APT is using the Filename field to craft HTTP GET request, the optimal accuracy should be 100%. We suspect therefore some mismatches due to the HTTP format.

The attack on size can be improved as well. Indeed, the optimal theoretical result are shown in section 5. By knowing more details about how APT resolves first-level dependencies, we are convinced that the attacker can get close to the optimal value. Moreover, we considered a threat model very restrictive, where the attacker sees only one flow of packages for possibly many package downloads (Fig. 4.2). This case happens if TLS is contacting only one Debian repository with a persistent connection. However, multiple flows appear when APT contact different Debian repositories. This allows the attacker to derive a better attack, since he will be able to identify multiple sizes of individual packages or group of packages.

REPRODUCIBILITY

All the code used for the practical and theoretical results are available publicly on Github at the following address: `https://github.com/alex35469/fingerpatch`.

Bibliography

[1] Justin Cappos. Justin Samuel Scott Baker John H. Hartman, 2008. *A Look In the Mirror: Attacks on Package Managers*, [online] Available at:<https://isis.poly.edu/~jcappos/papers/cappos_mirror_ccs_08.pdf> [Accessed 29 December 2018]

[2] The Horney, 2016. *A Comparison of Popular Linux Package Managers*. [online] Available at:<https://fusion809.github.io/comparison-of-package-managers/#toc6> [Accessed 31 December 2018]

[3] Debian Policy Manual v4.3.0.1 *7.1. Syntax of relationship fields*. [online] Available at:<https://www.debian.org/doc/debian-policy/ch-relationships.html> [Accessed 2 January 2019]

[4] Steve Ovens, 2018. *The evolution of package managers*. [online] Available at:<https://opensource.com/article/18/7/evolution-package-managers> [Accessed 20 December 2018]

[5] The Debian GNU/Linux FAQ. *7.9 - Basics of the Debian package management system*. [online] Available at:<https://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html#s-depends> [Accessed 29 December 2018]

[6] w3schools. *HTML URL Encoding Reference*. [online] Available at:<https://www.w3schools.com/tags/ref_urlencode.asps> [Accessed 25 December 2018]

[7] Tao Wang, Ian Goldberg, 2016. *On Realistically Attacking Tor with Website Fingerprinting*. Proceedings on Privacy Enhancing Technologies, pages 21 - 36.

Appendix

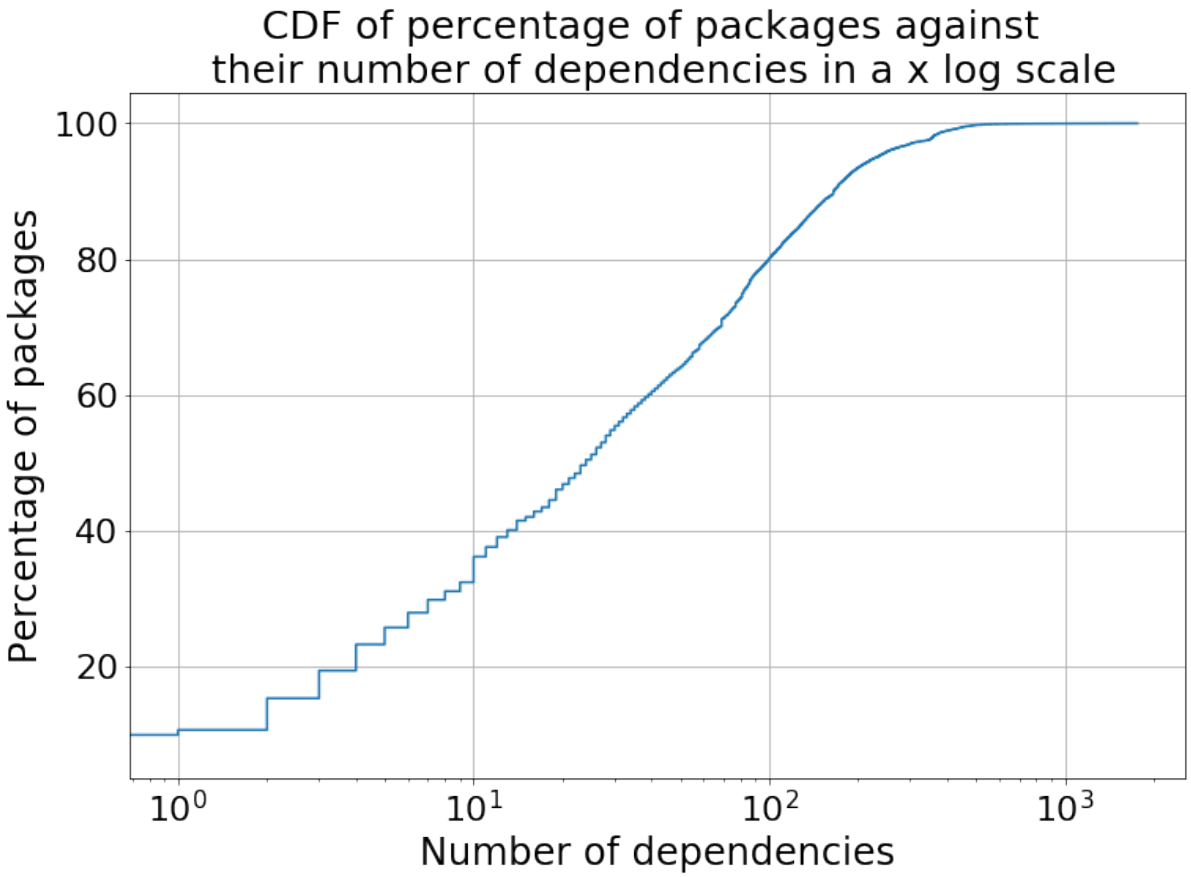

Figure 8.1: Cumulative distribution of the packages (in %) with respect to their number of dependencies on a log x scale