Poster: Classifying Downloaders

Yu Ding[1], Liang Guo[3],Chao Zhang[2], Yulong Zhang[2], Hui Xue[2], Tao Wei[2], Yuan Zhou[3], Xinhui Han[1]
Peking University[1], UC Berkeley[2] , Beihang University[3]
{dingelish,greenpear1991,gausszhch,ansonzyl,huixue.uiuc,lenx.wei}@gmail.com, zhouyuantd@163.com, hanxinhui@pku.edu.cn

***Abstract*—Downloader plays an important role in malware distribution, botnet construction and pay-per-install service. Downloader fetches specified programs from remote server and executes these programs under the attackers' control. Analyzing and classifying downloaders help us understand how downloaders propagate and evolve, also alert the system administrators about incoming attacks.**

I. Introduction

Downloader is a software which fetches software from remote server and executes the fetched files under the attackers' control. Not only botnet constructors, also software download sites, pay-per-install service providers utilize downloaders to spread their software. Downloader is in people's everyday life. People need to carefully click the links, in case of download malicious downloaders.

Analyzing downloader is different from analyzing traditional malware. It is because that downloader itself does not perform malicious behavior, such as stealing privacy, install backdoor etc. So behavior based malware analysis cannot apply effectively on downloader analysis. Also, downloaders are packed and encrypted in variety of ways. The packers and the encryptors make it hard for static analysis. What's more, some downloaders detect VM and resist on VM based analysis. This makes dynamic analysis harder and harder.

Previous researches on downloaders mainly focus on malware downloaders. Rossow et al. [1] characterizes malware downloaders on network level. AMICO [2] refines this research and shows more details about the traffic of malware downloaders. Nazca [3] is a novel approach to identify web requests related to malware downloads and installations. However these approaches characterize malware downloaders on network level. There is no deep analysis of general downloader binaries.

In this paper, we show DUST, our approach to automatically detect downloaders and cluster the detected downloaders. The input of DUST is binary files and the output is the cluster which the binary belongs to. DUST can be deployed together with gateway, which has the ability to collect executable binary files from network traffic. To conclude, our contributions are:

- We first look into the analysis of binary analyzed downloader clustering.
- We design and build DUST. DUST is an automated system which gets executables as input and output downloader cluster as result.

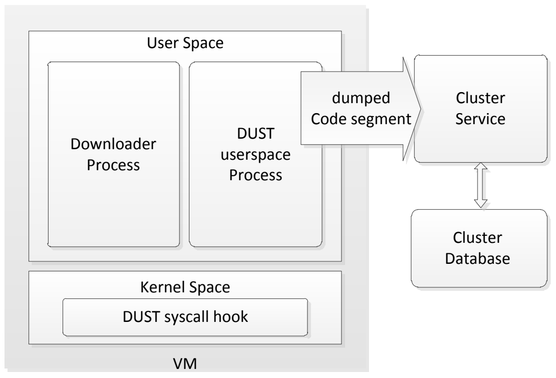

Fig. 1. **General Structure of DUST**

II. System Design

Figure 1 shows the structure of DUST. DUST runs input binaries in VMs and use system call hook at kernel level to monitor the behavior of input binaries. The userspace DUST process communicate with the kernel hook function and get essential runtime information about the process of input binary. Also, it judges whether the input binary is a downloader. If the input binary is judged as a downloader, DUST rolls back the VM and re-runs the binary. This time, the kernel level hook monitors all network activities and dump process' code section after the downloader unpack itself. And then the kernel level hook function passes the dumped code section to the user level DUST process and finally to the cluster service. The cluster service cluster the input code sections automatically and save the cluster information into the cluster database. The first run of input binary is called stage 1 and the second run of the input binary is called stage 2. The cluster procedure is stage 3.

A. Stage 1: Detect Downloader

In the first stage, DUST judge whether the input binary is a downloader. The name 'downloader' indicates that the input binary should download something and then execute it. So in the first stage, DUST judge whether the input binary acts out as a downloader.

We utilize dynamic taint analysis technique to achieve this goal. The taint source of this algorithm is incoming network bytes. We mark the network incoming bytes as tainted during the execution. The sink is file access API and process/thread create API. On file write, we mark the file contains tainted data as tainted file. On invoking `CreateProcess` series function, we check if the executed binary is a tainted file. On invoking `CreateThread` series function, we check if the code section at the thread entry point is tainted. If any execution starts on tainted bytes, DUST judges the input binary as downloader.

B. Stage 2: Unpack Downloader

In the first stage, the input binary is judged as downloader. Then DUST rolls back the VM and executes the input binary again to unpack it. In this stage, the kernel space hook function plays an important role of deciding when the downloader unpack itself. As is known to all, general unpack method is an unsolved problem. Here we assume that the unpack process does not access internet. The kernel system call hook function traces every system call of the input binary process and pause the process on first network access. Then it traces back the stack and locates the code section. In the end, it saves the code section (which invokes the network system call) to file and pass it to the user space DUST process.

C. Stage 3: Clustering

In stage 3, the userspace DUST process gets the unpacked code section and passes it to the cluster service. The cluster service gathers a certain amount of input code section and use a graph-based algorithm to cluster them.

III. DUST Implementation

The userspace DUST program for stage 1 is a dynamic taint analyzer. We port libdft [4] to Windows platform as dftwin [5]. On invoking of `NtDeviceIoControlFile` function, we check if it is a network input request. We mark the network input bytes as tainted. Also we check all the file writes in this system call and mark the written file which contains tainted bytes as tainted. On invoking of `NtCreateProcess` and `NtCreateThread` series of system call we check if the executed binary/code section contains tainted data. The system enters stage 2 if the user space DUST program detects downloader behavior in 10 minutes.

In stage 2, the kernel space hook system call by hooking SSDT table. It monitors every system call of `NtDeviceIoControlFile` and pause the process of input program on first network request. Then it traces back the user stack and locates the code section and dump the code section. The reason why we do not conduct such dump in stage 1 is because PIN has a special memory model and dynamically instrument codes in PIN code cache. We cannot precisely locate the unpacked codes in the first stage.

In stage 3, DUST cluster the dumped code sections. The cluster algorithm is based on CFG similarity calculation. The algorithm has three steps: (1) gather information of all basic blocks and functions, (2) create CFG for each function, (3) cluster the gathered code sections by compare the CFGs.

In (1), DUST uses IDA pro to decompile the dumped code section heuristically. Then it creates CFG for each dumped code section. The CFG includes how the basic block connects to each other in one function. Also the op code sequence in one basic block is saved. DUST saves all basic blocks into BB table in a database and indexes them using unique numbers.

In step (2), DUST creates CFG for each function. The CFG is described in adjacency lists. Once a CFG is created, DUST compares it with every CFG in CFG table in database by calculating the graph similarity [6] between them. If the newly generated CFG is different from all known CFGs, DUST inserts it to the CFG table together with the id of its code section. In such way, we create a table for all basic blocks and another CFG table to save all the CFGs of functions.

After (1) and (2), we have CFGs for each function and we index them by using code section id. In step (3), we compare the similarity between each pair of code sections. We use Jaccard index to evaluate the similarity of two code sections and create clusters.

IV. Preliminary Results

We collect 49732 binaries from a network gateway and detect 1161 of them performs downloader behavior (execute after download). The cluster service clusters the dumped code sections into 72 different clusters. The biggest cluster contains 126 samples. All the 126 samples are 32k-bytes and they falls into 10 different md5 hash values. The cluster algorithm successfully generated CFGs for them and found out that they are same to each other with similarity value over 99.9%. We compare our results with BitShred [7]. The accuracy is no less than the accuracy of BitShred.

V. Conclusion

In this paper, we show our design and implementation of DUST, a downloader detection and cluster tool. DUST uses dynamic taint analysis to judge if a binary is a downloader. DUST uses system call hook to dump unpacked downloader and cluster the downloaders by clustering the dumped code sections. Preliminary results show that DUST can detect software downloaders effectively.

References

[1] C. Rossow, C. Dietrich, and H. Bos, "Large-scale analysis of malware downloaders," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013, pp. 42–61.

[2] P. Vadrevu, B. Rahbarinia, R. Perdisci, K. Li, and M. Antonakakis, "Measuring and detecting malware downloads in live network traffic," in *ESORICS'13*, 2013, pp. 556–573.

[3] L. Invernizzi, S. Miskovic, R. Torres, S. Saha, S.-J. Lee, C. Kruegel, and G. Vigna, "Nazca: Detecting Malware Distribution in Large-Scale Networks," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, Feb 2014. [Online]. Available: http://seclab.cs.ucsb.edu/media/uploads/papers/invernizzi_nazca_ndss14.pdf

[4] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *In Procedeengs of the 19 th Network and Distributed System Security (NDSS)*, 2012.

[5] Y. Ding, "Dftwin, a dynamic taint flow analysis for windows." [Online]. Available: https://github.com/dingelish/dftwin

[6] S. Cesare and Y. Xiang, "Malware variant detection using similarity search over sets of control flow graphs," in *Proceedings of TrustCom'11*, Nov 2011, pp. 181–189.

[7] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011, pp. 309–320.