Avoiding Network and Host Detection using Packet Bit-masking

George Stergiopoulos[a], Eirini Lygerou, Nikolaos Tsalis, Dimitris Tomaras and Dimitris Gritzalis

Information Security & Critical Infrastructure Protection Laboratory, Department of Informatics, Athens University of Economics & Business, 76 Patission Ave., Athens GR-10434, Greece

Keywords: Network Security, Detection, Attack, Evasion, Intrusion Detection, Host, Siem, Malware, TCP, Packet, Transport, Layer, Payload, Shell, Data Leakage, DLP.

Abstract: Current host and network intrusion detection and prevention systems mainly use deep packet inspection, signature analysis and behavior analytics on traffic and relevant software to detect and prevent malicious activity. Solutions are applied on both system and network level. We present an evasion attack to remotely control a shell and/or exfiltrate sensitive data that manages to avoid most popular host and network intrusion techniques. The idea is to use legitimate traffic and victim-generated packets that belong to different contexts and reuse it to communicate malicious content without tampering their payload or other information (except destination IP). We name the technique "bit-masking". The attack seems able to exfiltrate any amount of data and execution time does not seem to affect detection rates. For proof, we develop the "Leaky-Faucet" software that allows us to (i) remotely control a reverse shell and (ii) transfer data unnoticed. The validation scope for the presented attack includes evading 5 popular NIDS, 8 of the most popular integrated end-point protection solutions and a Data Leakage Prevention system (DLP); both on the network and host session level. We present three different variations of the attack able to transfer (i) shell commands, (ii) large chunks of data, and (iii) malicious code to a remote command and control (CnC) center. During experiments, we also detected an NPcap library bug that allows resent packets to avoid logging from network analysis tools for Windows that use the Npcap library.

1 INTRODUCTION

There exists a variety of security measures for detecting and preventing malicious activity, with the purpose of both adhering to regulatory compliance and also reducing the risk in case of malicious activity. Security officers use Intrusion Detection Systems (IDS) (J., 1998), both at network level (NIDS) and host endpoint solutions (HIDS, heuristics, antivirus etc.) as both software applications and/or discrete hardware appliances. These systems monitor network connections and workstations for violations and the existence of malicious activity. (Marpaung et al., 2012).

On the other hand, researchers and technical experts keep publishing evasion attacks while others respond with more efficient detection techniques, both thus evolving the current state of the art with new ways to avoid measures and detect malicious activity.

Any malicious information (data, commands, etc.) is essentially a sequence of bits. The idea is to break this sequence into *k* bit blocks of length *n*, capture legitimate TCP/IP traffic from the victim's network and try to detect *k* TCP packet payloads (one for each bit block) where each packet has the same *n* bits in specific positions (in binary form) with a corresponding bit block. Essentially, we replace all *k* bit blocks of malicious data into *k* TCP packets of legitimate traffic, whose payloads contain the aforementioned bit blocks in some fixed bit positions. Fixed positions can be chosen arbitrarily by the attacker; similar to a symmetric encryption key. Lastly, we resend the captured packet to an attacker's listening service without tampering (expect of course changing the destination IP). Thus, we transfer a part of the overall intended malicious data that we want to send. We named this bit referencing technique, "bit-masking".

The bit-masking algorithm is effectively a mutation of a reverse-shell that connects a victim to its attacker. The difference is that a bit-masked connection uses existing legitimate traffic from the victim to transfer commands and data instead of creating new traffic. Payloads of legitimate TCP packets from victim-generated traffic are used as is to achieve

[a] https://orcid.org/0000-0002-5336-6765

Stergiopoulos, G., Lygerou, E., Tsalis, N., Tomaras, D. and Gritzalis, D.
Avoiding Network and Host Detection using Packet Bit-masking.
DOI: 10.5220/0009591500520063
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 52-63
ISBN: 978-989-758-446-6
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

all three attack scenarios presented in this paper. We develop a tool named Leaky-faucet as a proof of concept. The validation scope for the presented evasion attack includes detection mechanisms for malicious activity on the network and host session connection level (network and host IDS, endpoint security solutions, antivirus). Security mechanisms failed to detect Leaky-Faucet, both when executing commands and receiving output and exfiltrating data of any size, given enough time for the attack to run. We tested this for various data sizes from 40KB to 230MB files (of course the bigger the exfiltration, the longer the attack has to remain active).

Through the use of raw sockets, malicious commands and data are sent using entire legitimate payloads, instead of being inserted on existing sessions. This way, even if we check plain-text spoofed packets manually, the malicious content is undetectable.

1.1 Contribution

We present a fully functional reverse shell with the ability to hide commands and data transmitted over a victim's network. Our implementation takes advantage of raw payloads from legitimate TCP packets to bidirectionally transfer masked commands and data. Our major contributions are:

1. An evasion attack named bit-masking that manages to create a stable, reverse connection that none of the tested systems managed to detect. We send shell commands and receive output, transfer data and exfiltrate sensitive files in the order of tens of megabytes undetected.
2. We perform three attack scenarios to validate our attack performance in different implementations with numerous security measures. The attack was tested against 5 popular NIDS, 8 popular integrated endpoint protection solutions and a Data Leakage Prevention system; both on the network and host session level. Scenarios use (i) regular sockets, (ii) raw sockets and (iii) raw sockets with Npcap (Dean, 2016). If the NPCap library is installed on the victim's machine with default (admin) configuration, the Npcap Spoofed edition can be executed correctly from userland without administrator privileges.
3. During experiments, we also discovered a traffic monitoring bypass issue using the Npcap library. Specific attacks were able to avoid all session and packet logging from monitoring tools that use Npcap (e.g. Wireshark).

We should note that the goal of this paper is to evade detection of information exchange while connecting to the attacker's system at both the host and the network level. This paper does not focus on code injection and execution techniques like ROP injections, polymorphism and other techniques that aim to evade detection during code execution. We assume that a script is executed on the victim's machine using one of the numerous existing techniques and focus on detection evasion on network and host levels, both from userland and root.

Section 2 presents related work concerning detection of malicious activity both in network traffic and in host machines. We also argue about the differences with our presented attack. Section 3 describes the bit-masking evasion technique and its client-server model. Here we also present the three version of the attack, namely the TCP version, the spoofed raw and the spoofed NPcap version of the attack. Section 4 presents the various implementations of the attack (Leaky-Faucet) for remote command-and-control for all attack versions and our experiments to determine the most efficient bit-mask length, while Section 5 describes experimental results with remote commands and data leakage. Section 6 concludes and discusses potential solutions and future work.

2 RELATED WORK

2.1 Evasion Attacks

State-of-the-art evasion techniques include obfuscation, session fragmentation or splicing, application specific violations, protocol violations, inserting traffic at IDS, denial of service, and code reuse attacks (J., 1998).

Packet fragmentation is a network level evasion technique that utilizes the maximum transmission unit (MTU) of packets to generate malformed TCP payloads (Cheng et al., 2012; Martin, 2019). Splicing delivers malicious payloads over multiple network packets in a session through streams of small packets.

Other evasion techniques include desynchronizing protocol streams (Serna, 2002) and traffic injection attacks. Packet-based network intrusion detection systems can be avoided by making the IDS process different data or the same data differently than receiving systems in sessions (Bukac, 2010). Traffic injection attacks involves sending packets that will be processed by IDS but not by target machines, thus creating different sessions states between the IDS and target systems (J., 1998; Serna, 2002). A similar attack dubbed "duplicate insertion" uses overlapping segments from packets to confuse the IDS due to the system's ignorance on network topology.

Slow scans that fool the frequency checks of IDS by slowing down packets, method matching that uses alternate HTTP commands for detecting CGI scripts and premature request ending with malicious data hidden in headers are also evasion attacks frequently performed against the IDS (Martin, 2019).

Denial-of-service attacks have also been used for evading intrusion detection (Cheng et al., 2012) (Martin, 2019). These attacks try to overflow the network connection or the IDS system's resources to slow down rule matching or pattern recognition (Igure and Williams, 2008).

The evasion attack that is most similar to our approach is payload mutation. In this technique, attackers transform malicious packets to semantically equivalent that look different from malicious packet signatures (e.g. transformation of URI hexadecimal encoding, self-reference directories etc. in payloads) (Martin, 2019). Still, the aforementioned evasion attacks only work under certain cases and not in every situation (Cheng et al., 2012; Niemi et al., 2012). Contrary to this, we tested our approach against numerous, diverse types of security systems that implement different detection mechanisms (IDS, host endpoint security systems). Results show none of the tested systems could either detect the malicious connection or data and commands transferred over the network.

2.2 Malicious Activity and Data Leakage Detection Techniques

Most common host endpoint security mechanisms involve host intrusion detection systems (host IDS), antivirus and security solutions that utilize a range of techniques. From simple signature analysis and API call monitoring, sandboxing to more advanced detection mechanisms like packet caching, flow modification (Handley et al., 2001), active mapping (Shankar and Paxson, 2003) and policy enforcement (Marpaung et al., 2012; Cheng et al., 2012). The state of the art now also considers heuristics with machine learning.

Modern network security solutions use session packet heuristic analysis, deep packet inspection and session trends (like packets per min) along with botnet architectures to detect malicious activity in networks (Livadas et al., 2006; Binkley and Singh, 2006). Others rely on statistical analysis for classifying various types of traffic (Crotti, 2007), session reassembly (Martin, 2019) to detect splicing and sandboxing.

Some solutions use machine learning to detect malicious network activity. In (Lakhina et al., 2004) and (Stergiopoulos et al., 2018), researchers try various packet features to extract information from the physical aspects of the network traffic. Authors in (Prasse, 2017) use malicious HTTPS traffic to train neural networks and sequence classification to build a system capable of detecting malware traffic over encrypted connections. Other approaches focus on identifying target malware/botnet servers (Lokoc et al., 2016) or web servers contacted (Kohout and Pevny, 2015), instead of understanding malicious traffic of various types. Authors in (Gu et al., 2007) and (Yen and Reiter, 2008) use signal-processing techniques like Principal Component Analysis to aggregate traffic and detect anomalous changes flows. Lakhina et al. (Lakhina et al., 2004) modelled network flows as combinations of eigenflows to distinguish between short-lived traffic bursts, trends, noise, or normal traffic. Terrell et al. (Terrell, 2005) grouped network traces into time-series and selected features, such as the entropy of the packet and port numbers, to detect traffic anomalies.

Concerning data leakage, authors in (Tahboub and Saleh, 2014) surveyed DLP systems and described existing technologies for data protection, such as IDS/IPS, Firewalls, etc. that are using Deep Packet Inspection (DPI) architecture. They compared them to the DLP systems that use Deep Content Inspection (DCI) and showed how the latter are more efficient on detecting potential data breaches. Authors in (W"uchner and Pretschner,) presented a DLP solution based on Windows API function calls using function call interposition. In (Borders and Prakash,), researchers introduced an approach for quantifying information leaks in web traffic using measurement algorithms for the HTTP (Hypertext Transfer Protocol) protocol to isolate not legitimate outgoing activity.

While all aforementioned network detection and data leakage prevention approaches are based on models of malware behavior (not unlike signature-based intrusion detection), our approach uses completely different packets. This way, we can avoid detection by (a) not having to hide malicious information and (b) by introducing all types of payloads from different legitimate sessions in our malicious stream, thus data transferred cannot be assigned to a particular distribution nor can be analyzed based on static features.

2.3 Covert Channels and Steganography

The bit-mask is like a symmetrical encryption key. Its values are bit position pointers that need to be shared between the code running at the victim's and attacker's system prior to execution. If a mask is agreed on both sides (victim software and attacker's listener), each side can either pick-and-send or receive-and-decode TCP packets. Bit-masking attacks do not modify any bits on the victim packets nor do they inject bits into legitimate streams like in steganography.

The presented attack can be classified as a internet covert channel attack. Similar to Handel and Sandfor's data hiding analysis inside an OSI layer (Handel and Sandford, 1996), our technique also uses the structure of an existing layer (i.e. the TCP protocol and transport layer) to transfer information in an illicit manner, disguised as a legitimate stream. Contrary to most current implementation though, information is not hidden in TCP flags (ica,), but actually creates discrete connections or sends packets like any other network connection, while still avoiding being flagged as malicious by security measures or logged in network activity.

3 THE BIT-MASKING ATTACK METHODOLOGY

3.1 Bit-mask Definition

Essentially, a bit-mask is a one-dimensional vector of length μ that indicates some bit positions inside legitimate payloads, where μ is the amount of bit positions referenced by the bit-mask (i.e. mask length). For example, the bit-mask $\{5, 6, 16, 1, 19, 24\}$ with length $\mu = 6$, points to the *5th bit, 6th bit, 16th bit, ..., 24th bit* of a TCP *payload in binary form*.

A bit-mask is shared between the attacker's server and client code to be used as a bit reference manual for bit extraction. The attack model and algorithmic steps are presented below.

3.2 Attack Model

Here we present the high-level attack model for all following attack implementations. The model consists of a single attacker that has a client-side network program waiting for input. The server code is executed on a victim's workstation. Fig. 1 visualizes the high-level attack model.

The algorithm converts malicious data to binary form and breaks them down into k chunks of μ bits (the size of the bit mask). Then, it sniffs legitimate network traffic and detects k TCP packets from the ongoing legitimate traffic with payloads that have the same bits (in binary form) with the k malicious bit chunks, at the bit positions indicated by the bit mask. The algorithm then re-sends the legitimate packets to the attacker as is, by changing only the packet's destination IP. Packets are sent in sequential order. Bits indicated by the bit-mask are then extracted by the receiver and are concatenated with previous bits received from similar packets. The client code receives packets and appends the extracted bit contents indicated by the bit-mask to form the malicious data sent from the victim's system.

For the attack to be successful, the server code needs to achieve three things.

1. First, the victim needs to run the server code in the first place. Server code is less than 2KB and can exfiltrate any data or execute any command as if it was a shell. Execution needs either admin priviledges, or an existing install of the NPcap library with similar uses (e.g. a PC with Wireshark installed has this working library) If NPCap is present, the third attack requires no administrator priviledges. Various techniques such as droppers, malicious links to JavaScript or simple scripts can be used to achieve this; most of them are widely used. Since the aim of this paper is to present network detection avoidance attacks, we consider the way that the initial code is executed out of scope of this paper.
2. The second requirement is that the executed code must sniff enough packets from the victim's everyday traffic, to find payloads suitable to use with bit masking. As presented above, the attack does not modify victim packets.
3. The third requirement is for the client code to manage to send the selected packets to the attacker undetected by both client and network based detection systems (SIEMs, IDS, antivirus etc.).

The two latter requirements are presented below in subsections 3.2 and 3.3. If all requirements are met, the attacker has a shell and an upload/download terminal.

3.3 Sending Function - Victim Sniffing and Pattern Matching

The detailed steps of the code running on the victim's system are described below:

1. Convert malicious data D into binary and break them down to k blocks of bits of length μ, $D = k * \mu$.
2. Capture legitimate TCP/IP traffic, convert packet payloads to binary, and iterate them to detect packets that have the same bits with the bits from a block k_i, at positions indicated by the bit-mask.

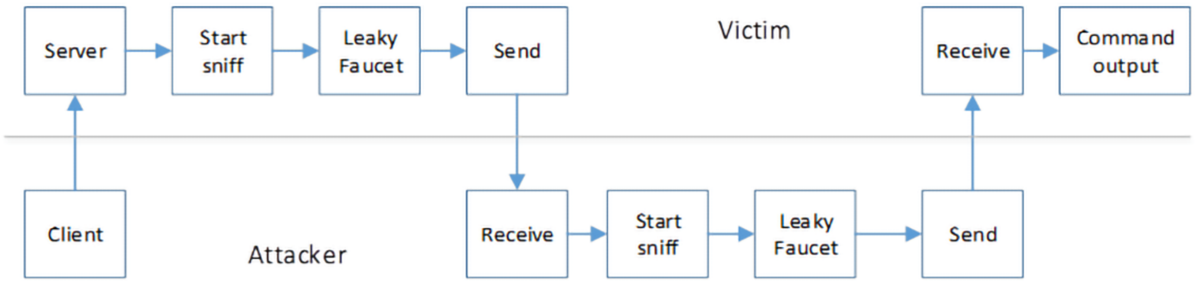


Figure 1: Activity flow of a full Leaky Faucet shell control.

3. If a suitable payload is found, replace the corresponding block of malicious data with the TCP packet.
4. Repeat until all malicious bit blocks are replaced with captured legitimate packets.
5. Resend captured legitimate packets sequentially to the attacker's IP.

A graphical representation is given below in Fig. 2 and Python code respectively below.

3.4 Receiving Function – Attacker

In the receiving function, we have the sniffing, pattern-matching and reassemble data operation. In this stage the attacker software:

1. Wait until all packets from the victim's system are received. Last packet is indicated by a bit-masked payload with a specific bit sequence that is used as an end mark.
2. When finished, code applies the mask at the first packet received and extracts the first chunk of bits from its binary payload, from the positions referenced by the mask.

   Repeat on all received packets in sequence.
3. When bit extraction finishes, concatenate k extracted bits blocks of length μ in the same byte array and
4. convert bits back to the original data's format (e.g. PDF, string for commands and output, hexadecimal for code etc.).

4 IMPLEMENTATIONS

4.1 Problem Definition

Each experiment has a client and a server. The server side is executed at the victim's system and the client side executes the attacker's software. We developed

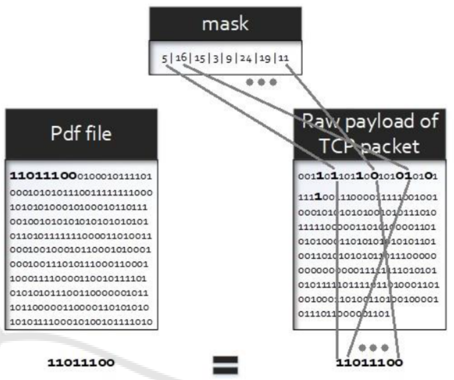


Figure 2: Using bitmask to detect a packet with bits identical to the bits from the file to be leaked.

three variations of the attack, each one with different socket and packet implementations to test detection by various security measures. All versions are essentially a reverse shell modified with the use of the bit-masking logic.

The code checks each packet as it is being collected and only keeps those packets that pass the check for being useful for the next chunk of data. Estimations show that we need on average 3.500 packets to detect one useful payload using an 8-bit mask, see Section 4.2 below for more details.

The reason we don't just keep outgoing packets is due to performance. We do not have any issues security wise, since we have been able to craft the source and the destination address (on the second and third edition), so keeping both incoming and outgoing packets increases our chances to find suitable payloads. Connection is made in reverse, from the victim to the attacker system since most security measures allow outgoing but block incoming new TCP connections.

To cope with packet loss, we opted to extend the bit-mask by 2 bits and use the latter two as a packet sequence number. When bits are to be matched, the last two bits are introduced as an incremental sequence number that resets and continues. If the receiving function detects erroneous sequence, it can notify the server code for the same packet again (again using he same bit-masking algorithm to communicate information back).

In the unlikely scenario where both server and client packets are lost (packet loss occurred both for a packet sent and the request to resend it), the attack will obviously fail, although this never happened during experiments.

4.2 Legitimate Traffic Generation

Traffic used to successfully execute bit-masking was random packets from everyday traffic from two workstations located inside the team's research lab. All users were aware of the ongoing sniffing for experimental purposes.

Workstation primary use was for coding and surfing. Traffic comprised of normal Windows and Linux everyday traffic (e.g. software updates and OS connections), together with user traffic (web browsing, streaming, YouTube music and Git push and pulls), on a 10/1Mbps download/upload connection.

During the sniffing part of the attack, workstations were used for surfing on web pages, coding on platforms (which introduced no extra traffic) and occasional streaming of music (which augmented the available traffic up to 11MB per minute). Also, default windows OS connections would occasionally run in the background, although no major updates were installed so the added network traffic is considered trivial (90MB per day). We did not notice any differences between encrypted and unencrypted connections.

Table 1: Traffic generation per user activity.

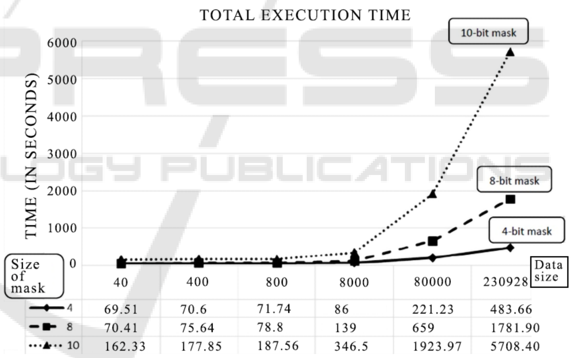
Traffic source	**Average load (MB/min)**
Web browsing	0.35
Youtube	9.34
Spotify (160kbps)	1.2
Git	0.75

4.3 Optimal Bit-mask Selection Tests

Masks can be arbitrary and attackers can may as well choose any mask they want. Preliminary tests on masks show that bits inside the mask play no important role; although the mask length greatly affects the time needed to deploy a successful attack.

Since the attack needs to find legitimate TCP payloads that have specific bits at the specific points referenced by our chosen mask, three variables affect how long the attack will take: (i) the size of the mask, (ii) the amount of legitimate traffic present and (iii) the amount of malicious information we want to transmit. For example, if we have an 8-bit mask and 8-bit length malicious data to send, then we need only one TCP packet that matches our mask. For every 8-bits or less that exceed the length of the mask we need a new packet. Smaller masks produce more collisions but utilize less data, so there is a trade off.

We ran tests to determine mask size and achieve the best results in the least amount of time, depending on the data size or/and the size of the mask. We tested all mask lengths up to 15. Figures 3, 4 and 5 depict examples for lengths 4,8 and 10. Anything above mask length of 10 takes too long to execute. We sampled mask sizes of 4, 8 and 10 bits on a 10/1Mbps download/upload private connection inside the team's research lab. Fig. 3 shows the time needed to execute a full attack that included utilizing a remote shell, creating a folder and downloading a sample NTLM sample file. Thwe *x* axis depicts different data sizes that we tried to exchange or exfiltrate (40KB, 400KB, 800KB, 8MB and 230MB files). Each size has a column that depicts that time (in seconds) needed to complete the attack for the aforementioned malicious data size.

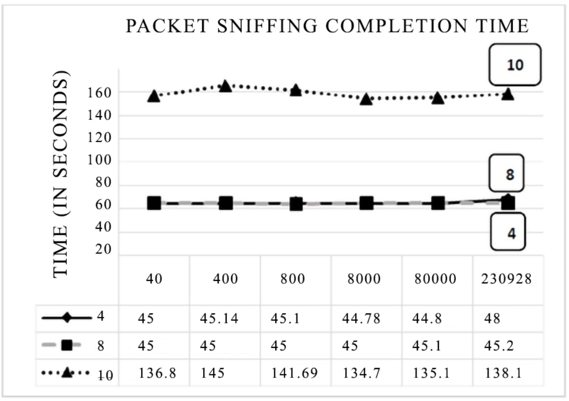


Size of mask	40	400	800	8000	80000	230928
4	69.51	70.6	71.74	86	221.23	483.66
8	70.41	75.64	78.8	139	659	1781.90
10	162.33	177.85	187.56	346.5	1923.97	5708.40

Figure 3: Total Execution Times of the executable for different sizes of mask and data.

Mask length tests show that, if an attacker wants to use bit-masking for scripting stuff and to establish the foothold, this attack is useful and fast. Still, for exfiltrating larger files (230MB), the bit-masking attack needed 8 minutes on a PC that was generating traffic equal to 15MB/min on average (all connections included).

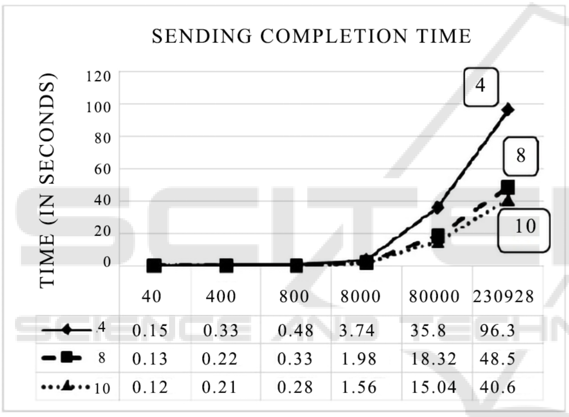
Graph functions below depict execution times (Fig. 4 and 5). Experiments suggest that using more than 10 bits for the bit-mask length, packet detection was getting significantly delayed. In Fig. 5, it is clear that going from a 4-bit to an 8-bit mask essentially halves the time needed to execute the attack for larger files. Still, the difference between an 8-bit and a 10-bit mask is almost non-existent. The possibility de-



	40	400	800	8000	80000	230928
4	45	45.14	45.1	44.78	44.8	48
8	45	45	45	45	45.1	45.2
10	136.8	145	141.69	134.7	135.1	138.1

Figure 4: Execution time for sniffing, for different sizes of mask and data.

creases exponentially. Tests show that using anything above a 10-bit mask slows down execution completion rather than speeding it up.



	40	400	800	8000	80000	230928
4	0.15	0.33	0.48	3.74	35.8	96.3
8	0.13	0.22	0.33	1.98	18.32	48.5
10	0.12	0.21	0.28	1.56	15.04	40.6

Figure 5: Execution time for sending completion procedure for different sizes of mask and data.

As we notice from charts above, the pattern matching and packet crafting procedures affect total execution time the most. Experiments show that 8-bit masks probably provide the best overall efficiency, except in the case of TCP packets with TLS Application Layer encryption. In TLS, the payload often starts with leading zeros (Group, 2018), and thus the mask must either contain bigger numbers or we omit the leading zeros. We chose to implement the bit-masking evasion attack on the TCP protocol since it is easy to implement connection that transfer raw packets; something we need for our second and third experiments. The average time for a packet to be matched and crafted to be ready for sending is depicted in Table 1.

Table 2: Average time for packet matching and crafting.

	4-bit mask	**8-bit mask**	**10-bit mask**
Time (sec)	0.0075	0.0624	0.2414
Speed (patterns/sec)	170.9	15.9	4.1

4.4 Implementation 1: TCP Version

In this simplest form of the attack, we utilize raw sockets for sniffing packets to match and TCP sockets to send information to the attacker. New TCP packets are created for every send and receive and we only use the captured payload from the previously captured packets. Readers should note that, of course, since a handshake is occurring with the attacker, we could simply encrypt the communicated bit strings. Still, we present this simple implementation as a *stepping stone* for the next attack variations, building up to the third implementation which is the most sophisticated one and eventually *compare results* from all three implementations.

This is the first PoC implementation and, in this stage, admin rights are needed for sniffing (the third implementation will remove this constraint). No additional restrictions exist. An active, connected socket is bound to an attacker's client with a particular IP address and port number (Rhodes and Goerzen,). We use a known TCP port to draw less attention. Network card is set in promiscuous mode. Every socket sends a single packet (a condition required to send data in order).

On the attacker's machine, sniffing starts from Ethernet frames (Data link Layer 2) (Standardization, 1996). On the victim's machine, sniffing starts from the Data link Layer 3 due to raw sockets limitations for Windows systems (Heuschkel et al., 2017) and also since most firewalls limit their restrictions for outgoing packets. We had to manually filter the packets with the *proto* label in the IP header, because the sniffing for IPPROTO_TCP is not permitted.

It is important that packets arrive in order. For this reason, we created new connections for every packet sent by the victim to the attacker's machine. We exploited a socket feature that blocks the socket until a packet is received. This way, the victim software did not send an END signal and timeout was triggered after the last packet.

4.5 Implementation 2: Spoofed Raw Version

In this version, we opted to test the use of raw sockets and see what will happen. The difference with implementation 1 is that, instead of crafting new packets and inserting selected payloads, we keep the original legitimate packet by the victim (same TCP header) and simply change its IP address at the network layer to match the attacker's IP. In this implementation, administrator privileges are required. In addition, most commercial firewall settings only allow outgoing traffic to be captured in sniffing mode which might cause delays.

Sniffing was performed similarly to the first implementation. Matched packets collected from the sniffing function are sent as is, except for the spoofed IP destination address. The Ethernet and IP headers are added automatically when we send the packet. The socket.IPPROTO_RAW option allows direct interaction with layer 3 (IP) to craft our custom packet with the default IPV4 IP protocol.

Without the TCP protocol, transmission of the packets is not reliable, although we did not get any misses. On the server side, no external library is used. This implementation gave us full control of a Windows or Linux shell with privileges identical to the ones owned by our victim's python scripts.

4.6 Implementation 3: Spoofed Npcap Version

In this version, we made use of the Npcap library (Dean, 2016) on the victim's side to be able to sniff the traffic behind the firewall. The library binded directly to the device interface and thus, we overcame limitations of the previous raw sockets in Windows. This version is similar to the spoofed raw version, but with the added advantage of not needing raw sockets server side (Dean, 2016). To access the library we directly used the Winpcap python wrapper. This way, the attack now required no privilege escalation for all executions and communication and everything ran smoothly from userland. If NPCap is installed on the victim's machine with default (admin) configuration, this attack requires no elenated priviledges and can be executed from userland.

5 EXPERIMENTS

The proposed system was tested on a Dell Inspiron 15-3537 (Intel Core i7-4500U, 16GB RAM). Leaky Faucet's code was written in Python version 3.7.2. The first two editions have no external dependencies and use default python libraries. The third variations of Leaky Faucet uses the NPcap library.

We set up an intranet lab using Oracle's VirtualBox 5.2.26 and Cisco routers as depicted in Fig. 6. Virtual machines formed a network were one Kali Linux machine represented the attacker and multiple MS Windows machines that represented the victims with Windows 7 64-bit and Windows 10 64-bit systems running different host endpoint security suites, antivirus, IDS etc.

We used the Security onion (Onion, 2019) distribution for implementing and configuring numerous NIDS (see Section 5.1, Table 3). We also tested 8 of the most popular integrated endpoint protection solutions (see Section 5.2, Table 4 below) and two Data Leakage Prevention systems (DLPs). We ran multiple shell commands to test functionality, such as "net user" commands, deleting and creating new folders, files, altering users, registry entries etc.

Section 5.1 and 5.2 below depict results for network IDS and host endpoint security systems respectively.

5.1 IDS and Network Security Evasion

The network intrusion detection and monitoring systems tested were the following:

- **Suricata.** Suricata is capable of real time intrusion detection (IDS), inline intrusion prevention (IPS), network security monitoring and offline processing (Foundation, 2019).
- **Snort.** A Network Intrusion Detection System and Intrusion Prevention System (Roesch, 1999). Performs real-time traffic analysis and packet logging, and is able to stop a number of probes and attacks on a network.
- **Bro.** Network Security Monitor (Paxson, 1999) that has the ability to detect events in real time. These events are by default neutral and signify that a notable action has taken place, regardless of that action's nature.
- **Ossec.** OSSEC (Bray et al., 2008) is a multiplatform, open source, host-based Intrusion Detection System (HIDS). It has a correlation and analysis engine, integrating log analysis, file integrity checking, Windows registry monitoring, centralized policy enforcement, rootkit detection, real-time alerting and active response.
- **Sguil.** (Visscher and Viklund, 2013) a tool that uses a number of the installed utilities to collect,

analyze and escalate indications and warnings to detect and respond to possible intrusions.

- **Pfsense.** (Buechler and Pingle, 2009) an open source suite based on FreeBSD that supports a number of features present in alternative commercial firewall solutions including filtering based on IP and Protocol, passive OS/network fingerprinting and packet normalization.

Security Onion is a suite that provides popular IDS implementations. For the Snort setup we used the Snort VRT ruleset and for the Bro, its default rules. The PFsense firewall used manages the internet access through the VMs. The virtual machines are in an internal network and can only see each other which provides unbiased network traffic. Fig. 6 below visualizes the network topology of our experiment.

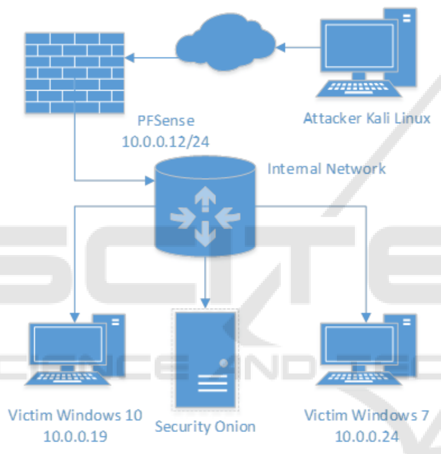


Figure 6: Network topology with IDS.

In the following Table 2, we present the comparison between the three editions with the installed Antivirus products and the network IDS.

Table 3: Detection results - NIDS.

NIDS - NIPS	*TCP*	*Spoofed raw*	*Spoofed NPCAP*
Snort	warning	*pass*	*pass*
Zeek (Bro)	Logged (no warning)	Logged (weird.log)	*pass*
Suricata	Warning (unknown)	Warning (unknown)	*pass*
OSSEC	*pass*	*pass*	*pass*

For Zeek, the Spoofed RAW variation of the attack raised an entry in weird.log (Zeek stores unusual network behavior in this log) with “255 unknown protocol entry”. It did not provide any further information, but we know that IPPROTO_RAW socket protocol in windows has a 255 number. Consequently, the log occurs when we send the packets from the server to the client using the IPPROTO_RAW socket. Suricata provided similar results. For the Npcap Spoofed Version no logs or alerts were detected. Using this technique, malicious packets always appeared legitimate.

Since packets selected by these techniques largely belong to different contexts (i.e., fragments from different legitimate flows), we opted to see if we can detect them using deep content inspection. Basic functionality tested did not provide any detection results although this area needs further research to claim that our attacks remains undetected.

5.2 Host Endpoint Security Evasion

In this experiment, we tested the three variations of the attack against host endpoint protection, host IDS and business antivirus suites and compared the results. We selected them based on various internet rankings from companies and press alike (Williams, ; Haselhorst, 2019). When there was more than one version, we installed both the Total Security (host IDS) and the antivirus versions. Detection results for the bit-masking attack against host endpoint security solutions are shown in Table 4 below.

The high rate of blocking in the Spoofed version, is due to the raw socket limitations on Windows. Most endpoint systems of most companies implement more flexible firewall rules instead of blocking all incoming connections by default, thus our attack can pass without any problems. We have no detections ratings, since workstation security products don’t provide statistics of behavior analysis. Besides ZoneAlarm PRO, no HIDS or host endpoint security solution detected or blocked our attack. The Kaspersky HIDS Application Control system classified Leaky Faucet at the “Low Restricted” trust category. Kaspersky Total Security did not block the execution of the application nor the packet sending procedure, although the total time needed for the attack to be executed completely increased by 58.971 seconds (+ 3.31%) when leaking megabytes of data. The Zone Alarm PRO’s firewall blocked our attack since it has very strict whitelisting firewall policies, disables promiscuous mode by default and as a result blocks our attack although does not detect it.

Our first edition passed all installed systems (even Zone Alarm since it doesn’t need promiscuous mode and doesn’t use raw packets), even though it created new TCP packets with the payload of the legitimate ones and established numerous connections to send

Table 4: Detection results - Host endpoint security.

Host endpoint security system	*TCP*	*Spoofed Raw*	*Spoofed NPCAP*
Windows Defender	pass	blocked	pass
McAfee AntiVirus Plus	pass	blocked	pass
McAfee Total Protection	pass	pass	pass
Symantec Norton Security Premium	pass	pass	pass
Kaspersky Internet Security	pass	pass	pass
ESET NOD32 Antivirus	pass	blocked	pass
Trend Micro Internet Security	pass	pass	pass
ZoneAlarm PRO Antivirus + Firewall	pass	blocked	blocked

or receive packets. Still, in a real world scenario, advanced endpoint solutions and SOCs would probably raise some warning flags since the attacker's IP is not hidden and all those multiple reconnections would be suspicious. Our third edition is the best choice for most systems, because it captures the data before any firewall can block traffic and avoids logging from NPCap tools.

5.3 DLP Evasion

In this experiment, we tested two variations of data leakage prevention with MyDLP (MyD, 2020). We set up an Ubuntu Server VM and used it as a proxy through Squid, a caching and forwarding HTTP web proxy, to monitor network traffic of the test VM (Windows 10). For testing, we used a PDF file with credit card data and we tried to send it through http://wetransfer.com. MyDLP recognized that sensitive data were being sent and logged the activity as configured to do in such case.

Using Leaky Faucet, MyDLP did not log any suspicious activity on the network, which means that it did not recognize the patterns used for the leak of the credit card numbers.

The total time needed for Leaky Faucet to send the file increased by 782 seconds compared to sending the file through WeTransfer, when MyDLP was logging the traffic.

5.4 Session and Packet Monitoring Bypass with NPcap

Wireshark and most popular Windows capturing tools use the NPcap library (or WinPcap) to capture packets when firewalls and other host systems are in use.

Without the use of NPcap, firewall blocks packets either way and cannot be sniffed. However, since we use the same library, we noticed that when we send the crafted new packets from the Victim to the Attacker, the library cannot simultaneously inject them and display them on the same machine on other monitoring tools. We tried to change Ethernet source and IP Source to Wireshark and still, no software displays them. We also used WindDump (tcpDump) and, again, capturing did not display any injected packets. Intuitively, if a monitoring tool utilizes its own implementation of the NPcap library, monitoring will possibly work since there will be no conflict.

Most likely, this is a library issue, because in Linux, when tested with raw sockets, the same monitoring tools can display packets injected by Leaky Faucet version 3 normally. Still, we found no tool able to display injected packets in Windows.

5.5 Whitelisting Network Connections

The most effective (and probably the simplest) solution to the bit-masking attack is whitelisting outgoing connections and checking the validity of software connecting to external addresses. This is a common and effective countermeasure against malware and reverse shells in general, and bit-masking attacks are also affected by it. Still, many industrial environments or organisation workstations use blacklisting instead of whitelisting connections, due to the inherent nature of company business logic that some employees need access to the internet in general. To this end, whitelisting and firewall restrictions on outgoing connections will stop bit-masking attacks, but it is a situational security measure that is not always possible to deploy.

6 CONCLUSIONS

In this paper, we presented an evasion attack able to bypass network, and host intrusion detection and data leakage prevention systems using a technique we named "bit-masking". The third edition has better performance due to the significant advantage of the Npcap library and the ability to capture packets before any firewall policy is applied and thus successfully perform packet injection.

Spoofing the IP and using raw sockets is worse than just creating a new TCP connection and embedding previous, legitimate payloads inside the new packets. Although the attacker needs to establish multiple connections, still the connection looks more legitimate than raw, unintended packets being send over the network; better use full TCP handshakes.

Using the NPCAP library in Windows tests provided interesting side effects. Monitoring tools cannot display the injected packets due to some conflict of the NPcap library being used both by Leaky Faucet and the monitoring tools like Wireshark. Linux seems to be unaffected of this issue, which probably is a library bug.

From a defense point of view, future work should aim at understanding context relevance in series of packets. DPI and relevant machine learning algorithms may possibly be able to detect this attack by understanding the completely different structure and data type in different packets (since each packet comes from different types of white traffic).

6.1 Restrictions

Each implementation deals with loss of synchronization (e.g. due to packet drops) between client code and server (victim) code differently. The first implementation uses full handshakes, so each socket connection transfers one packet; a blunt but effective first step to test detection with multiple TCP handshakes. in the second and third implementations, we opt to embed a two binary digit sequence number, much like TCP's initial sequence number (ISN) to check for sequencing. This number is concatenated to data sent with the bit-mask. Results so negligible time differences and no detection differences when using this technique.

Also, the attack will have serious delays if a victim's network traffic is minimal. Tests show that, in workstations with less than 20MBs of traffic per hour (i.e. idle PCs), the attacks take an estimate of five times more time to execute. For example, the shell attack that needs to transfer about 8MB of traffic to fully execute all commands and exfiltrate all data, takes about 5 hours to complete (instead of 86 seconds with the tested 10MBps traffic).

6.2 Future Work

Concerning the attack, future work must aim to build a more scalable version of the attack; able to leak larger files with greater speed, possibly by enhancing the payload matching algorithm. We also aim to extend the system to incorporate more features like encryption and TLS features to further enhance the evasion mechanisms.

REFERENCES

What is an internet covert channel? https://www.icann.org/news/blog/what-is-an-internet-covert-channel. Accessed: 2019-10-30.

(2020). Mydlp and security features. http://www.mydlp.com/. Accessed: 2019-09-30.

Binkley, J. and Singh, S. (2006). An algorithm for anomaly-based botnet detection. In *Proceedings of the Workshop on Steps to Reducing Unwanted Traffic on the Internet*.

Borders, K. and Prakash, A. *Quantifying Information Leaks in Outbound Web Traffic*. Web Tap Security Inc., Ann Arbor, MI.

Bray, R., Cid, D., and Hay, A. (2008). *OSSEC host-based intrusion detection guide*. Syngress.

Buechler, C. M. and Pingle, J. (2009). *pfsense: The definitive guide*. Reed Media Services.

Bukac, V. (2010). *IDS system evasion techniques*. Master. Masarykova Univerzita.

Cheng, T.-H. et al. (2012). Evasion techniques: Sneaking through your intrusion detection/prevention systems. *IEEE Communications Surveys & Tutorials*, 14(4):1011–1020.

Crotti, M. e. a. (2007). Traffic classification through simple statistical fingerprinting. *ACM SIG- COMM Comput. Commun. Rev*, 37(1):5–16.

Dean, P. (2016). Nmap security scanner gets new scripts, performance boosts.

Foundation, O. I. S. (2019). Suricata.

Group, N. W. (2018). The transport layer security (tls) protocol version 1.3.

Gu, G., Porras, P., Yegneswaran, V., Fong, M. W., and Lee, W. (2007). Bothunter: detecting malware infection through ids-driven dialog correlation. In *Proceedings of the USENIX Security Symposium*.

Handel, T. G. and Sandford, M. T. (1996). Hiding data in the osi network model. In *International Workshop on Information Hiding*, pages 23–38. Springer.

Handley, M., Paxson, V., and Kreibich, C. (2001). Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc.* USENIX Security Symposium.

Haselhorst, D. (2019). *Onion-Zeek-RITA: Improving Network Visibility and Detecting C2 Activity*. SANS reading room, sti graduate student research edition.

Heuschkel, J. et al. (2017). Introduction to raw-sockets.

Igure, V. M. and Williams, R. D. (2008). Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Commun. Surveys Tutorials*, 10(1):6–19.

J., C. (1998). Proceedings of the 1998 national information systems security conference (nissc 98). In *Artificial neural networks for misuse detection*.

Kohout, J. and Pevny, T. (2015). Automatic discovery of web servers hosting similar applications. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 1310–1315. IEEE.

Lakhina, A., Papagiannaki, K., and Crovella, M. (2004). Structural analysis of network traffic flows. In *Proceedings of ACM SIGMETRICS/Performance*.

Livadas, C., Walsh, B., Lapsley, D., and Strayer, T. (2006). Using machine learning techniques to identify botnet traffic. In *Proceedings of the IEEE LCN Workshop on Network Security*.

Lokoc, J., Kohout, J., Cech, P., Skopal, T., and Pevy, T. (2016). k-nn classification of malware in https traffic using the metric space approach. In M., W. and G. A., C. a., editors, *Chau*, pages 131–145. Springer, Cham, PAISI 2016. LNCS, vol. 9650.

Marpaung, J. A. P., Sain, M., and Lee, H.-J. (2012). Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*. IEEE.

Martin, S. (2019). *Anti-IDS tools and tactics*. SANS institute.

Niemi, O.-P., Levom"aki, A., and Manner, J. (2012). Dismantling intrusion prevention systems. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications*, architectures, and protocols for computer communication. ACM. technologies.

Onion, S. (2019).

Paxson, V. (1999). Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23):2435–2463.

Prasse, P. e. a. (2017). *Malware Detection by HTTPS Traffic Analysis*.

Rhodes, B. and Goerzen, J. Foundations of python network programming. *Apress*, 2014.

Roesch, M. (1999). Snort: Lightweight intrusion detection for networks. 99(1).

Serna, F. J. (2002). *Polymorphic shellcodes vs. application IDSs*. Next Generation Security Technologies.

Shankar, U. and Paxson, V. (2003). Active mapping: Resisting nids evasion without alerting traffic. In *Proc.* IEEE Symposium on Security and Privacy.

Standardization, I. (1996). Iso/iec 7498-1: 1994 information technology - open systems interconnection - basic reference model: The basic model. *International Standard ISOIEC*, 7498:59.

Stergiopoulos, G. et al. (2018). Automatic detection of various malicious traffic using side channel features on tcp packets. In *European Symposium on Research in Computer Security. Springer, Cham.*

Tahboub, R. and Saleh, Y. (2014). Data leakage/loss prevention systems (dlp). In *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, pages 1–6. IEEE.

Terrell, J. e. a. (2005). Multivariate svd analyses for network anomaly detection. In *Proceeding of ACM SIGCOMM*.

Visscher, B. and Viklund, A. (2013). Sguil: The analyst console for network security monitoring.

Williams, M. *The best antivirus software 2019*. TechRadar.

W"uchner, T. and Pretschner, A. *Data Loss Prevention based on data-driven Usage Control*. Technische Universit"at M"unchen, Garching bei M"unchen, Germany.

Yen, T.-F. and Reiter, M. K. (2008). Traffic aggregation for malware detection. In Zamboni, D., editor, *DIMVA 2008, vol. 5137*, pages 207–227. Springer LNCS, Springer, Heidelberg.