SCIENTIFIC and TECHNOLOGICAL COOPERATION

between

RTD ORGANISATIONS in GREECE

and RTD ORGANISATIONS in U.S.A, CANADA, AUSTRALIA, NEW ZEALAND, JAPAN, SOUTH KOREA, TAIWAN, MALAISIA and SINGAPORE

HELLENIC REPUBLIC

MINISTRY OF DEVELOPMENT

GENERAL SECRETARIAT FOR RESEARCH & TECHNOLOGY

International S & T Cooperation Directorate, Bilateral Relations Division

Project MILTIADES: **M**ult**I**-**L**ayer **T**echn**I**ques for **A**ttack **DE**tection **S**ystems

Deliverable 3.1: System Implementation

This document presents an overview of the system implementation and the results of our prototypes. In particular, in this document, we elaborate on the design and implementation of *nemu*, a malware detector operating on the network level and we discuss two approaches for implementing address space randomization.

Due Delivery Date	30/09/2007
Actual Delivery Date	15/11/2007
Participants	FORTH-ICS, Virtual Trip, Columbia University

Project Partners

FORTH-ICS	Project Leader	Greece
Columbia University	Project Leader	US
Virtual Trip	Co-operating organisation	Greece

Table of Contents

1. Introduction 3
2. Emulation-based polymorphic attack detection 4
   a. Implementation 4
   b. Performance Evaluation 4
   c. Tuning the Detection Heuristic 5
   d. Validation 6
   e. Processing Cost 8
3. Address Space Randomization 11
   a. DHCP-based implementation 11
   b. Transparent Address Obfuscation 12
   c. Simulation study 15
4. Conclusions 16
5. References 17

1. Introduction

The increasing number of malware observed daily and the advanced complexity of cyberattacks has emerged the need for fast and accurate defense. Current trends of threat landscape include polymorphic attacks and hitlist worms. Polymorphic attacks mutate their attack vector in such a way that each instance is completely different from others. Traditional defense mechanisms are unable to cope with polymorphism as they are based on identification of well-known signatures or employ heuristics for detecting anomalies in network traffic. Both approaches suffer from the problem of vast amounts of false positives and false negatives. Hitlist worms precompute their victims using secondary mechanisms, such as web search engines or peer-to-peer networks, forming a list of vulnerable population. At the time of attack launch, only the victims of this list are affected. Worm detection techniques that are based on content prevalence and connection characteristics of spreading are rendered useless against hitlist-based worms.

In our effort to defend against polymorphic attacks and hitlist worms, we propose two systems. The first one deals with the detection of polymorphic attacks and is based on the idea that data directed to a server should not include executable code. The technique used is emulation-based, that is network data are treated as assembly code. If a block of executable code is detected inside the network stream, then we have a strong indication for an ongoing attack. The advantages of this approach are the low false positive rate and its ability to detect previously unknown attacks, also called 0-day attacks. To prevent from hitlist worms, network address space randomization (NASR) is proposed. The goal of NASR is to make vulnerable population become a moving target and thus allow it escape its entry to the hitlist. Hosts change their IP address frequently enough so as the information gathered by the attacker is rendered stale when the attack is launched.

This document describes the implementation and performance aspects of both approaches. The rest of the document is organized as follows. In Section 2 we describe the implementation and tradeoff of emulation-based polymorphic attack detection. Section 3 provides two possible implementations of NASR and provides a picture of its imposed network overhead. We conclude in Section 4.

2. Emulation-based polymorphic attack detection

Emulation-based polymorphic attack detection is based on the actual execution of attack data on a CPU emulator. Our approach does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks. The main principle of the approach is the use of a generic heuristic that matches the runtime behavior of polymorphic shellcodes. At the same time, the actual execution of the attack code on a CPU emulator makes it robust to evasion techniques such as highly obfuscated or self-modifying code.

a. Implementation

The detector passively captures network packets using libpcap [10] and reassembles TCP/IP streams using libnids [11]. The input buffer size is set to 64KB, which is enough for typical service requests. Especially for web traffic, HTTP/1.1 pipelined requests are split to separate streams, otherwise an attacker could evade detection by filling the stream with benign requests until exceeding the buffer size. Instruction set simulation has been implemented interpretively with a typical fetch, decode, and execute cycle. Instruction decoding is performed using libdasm [12]. For our prototype, we have implemented a subset of the IA-32 instruction set, including most general-purpose instructions, but no FPU, MMX, SSE, or SSE2 instructions, except fstenv/fnstenv, fsave/fnsave, and rdtsc. However, *all* instructions are fully decoded, and if an unimplemented instruction is encountered, the emulator proceeds normally to the next instruction. The implemented subset suffices for the complete execution of all tested shellcodes. Even the highly obfuscated shellcodes generated by the TAPiON engine [13], which intersperses FPU instructions among the decoder code, are executed correctly, since any FPU instructions are used as NOPs and do not take part in the useful computations of the decoder.

b. Performance Evaluation

We evaluate the performance of the proposed approach using our prototype implementation. In all experiments, the detector was running on a PC equipped with a 2.53 GHz Pentium 4 processor and 1GB RAM, running Debian Linux (kernel v2.6.7). For trace-driven experiments, we used full packet traces of traffic from ports related to the most exploited vulnerabilities, captured at ICS-FORTH and the University of Crete. Trace details are summarized in Table 1. Since remote code-injection attacks are performed using a specially crafted request to a vulnerable service, we keep only the client-to-server traffic of network flows. For large incoming TCP streams, e.g., due to a file upload, we keep only the first 64KB. Note that these traces represent a significantly smaller portion of the total traffic that passed by through the monitored links during the monitoring period, since we keep only the client-initiated traffic.

Table 1 Characteristics of client-to-server network traffic traces

Service	Port Number	Number of Streams	Total size
www	80	1759950	1.72 GB
NetBIOS	137-139	246888	311 MB
Microsoft-ds	445	663064	912 MB

c. Tuning the Detection Heuristic

We first assess the possibility of incorrectly detecting benign requests as polymorphic shellcode. The detection criterion requires the execution of some form of getPC code, followed by a number of payload reads that exceed a certain threshold. Our initial implementation of this heuristic was the following: if an execution chain contains a call, fstenv, or fsave instruction, followed by PRT or more payload reads, then it belongs to a polymorphic shellcode. The existence of one of the four call, two fstenv, or two fsave instructions of the IA-32 instruction set serves as an indication of the potential execution of getPC code. We evaluated this heuristic using the traces presented in Table 1 as input to the detection algorithm. Only 13 streams were found to contain an execution chain with a call or fstenv instruction followed by payload reads, and all of them had non-ASCII content. In the worst case, there were five payload reads, allowing for a minimum value for PRT = 6. However, since the false positive rate is a crucial factor for the applicability of our detection method, we further explored the quality of the detection heuristic using a significantly larger data set.

Table 2 Streams that matched the detection heuristic with a given number of payload reads

Payload Reads	Streams			
	Initial Heuristic		Improved Heuristic	
	#	%	#	%
1	409	0.02045	22	0.00110
2	39	0.00195	5	0.00025
3	10	0.00050	3	0.00015
4	9	0.00045	1	0.00005
5	3	0.00015	1	0.00005
6	5	0.00025	1	0.00005
7-100	44	0.00220	**0**	**0**
100-416	37	0.00185	**0**	**0**

We generated two million streams of varying sizes uniformly distributed between 512 bytes and 64KB with random binary content. From our experience, binary data is much more likely to give false positives than ASCII only data. The total size of the data set was 61 GB. The results of the evaluation are presented in Table 2, under the column "Initial Heuristic." From the two million streams, 556 had an execution chain that contained a getPC instruction followed by payload reads. There were 44 streams with tens of payload reads and 37 streams with more than 100 payload reads, reaching 416 in the most extreme case. There are polymorphic shellcodes that execute as few as 32 payload reads. As a result, PRT cannot be set to a value greater than 32 since it would otherwise miss some polymorphic shellcodes. Thus, the above heuristic incorrectly identifies these cases as polymorphic shellcodes. Although only the 0.00405% of the total streams resulted to a false positive, we can devise an even more strict criterion to further lower the false positive rate. Payload reads occur in random code whenever the memory operand of an instruction *accidentally* refers to a location within the input buffer. In contrast, the decoder of a polymorphic shellcode explicitly refers to the memory region of the encrypted payload based on the value of the instruction pointer that is pushed in the stack by a call instruction, or stored in the memory location specified in an fstenv instruction. Thus, after the execution of such an instruction, the next mandatory step of a getPC code is to read the instruction pointer from the memory location where it was stored. This led us to further enhance the detection criterion as follows: *if an execution chain contains a* call*,* fstenv*, or* fsave *instruction, followed by a read from the memory location wherethe instruction pointer was stored as a result of one of the above instructions, followed by PRT or more payload reads, then it belongs to a polymorphic shellcode.* Using the same data set, the enhanced criterion results to significantly fewer matching streams, as shown under the column "Enhanced Heuristic" of Table 2. In the worst case, one stream had an execution chain with a call instruction, an accidental read from the memory location of the stack where the return address was pushed, and six payload reads, which allows for a lower bound for PRT = 7.

d. Validation

Polymorphic Shellcode Execution. We tested the capability of the emulator to correctly execute polymorphic shellcodes using real samples produced by off-the-shelf polymorphic shellcode engines.We generated mutations of an 128 byte shellcode using the Clet [15], ADMmutate [14], and TAPiON [13] polymorphic shellcode engines, and the Alpha2 [16], Countdown, JmpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShigataGaNai shellcode encryption engines of the Metasploit Framework [17]. For each engine, we generated 1000 instances of the original shellcode.

Figure 1 shows the average number of executed instructions that are required for the complete decryption of the payload for the 1000 samples of each engine. The ends of range bars, where applicable, correspond to the samples with the minimum and maximum number of executed instructions. In all cases, the emulator decrypts the

original shellcode correctly. Figure 2 shows the average number of payload reads for the same experiment. For simple encryption engines, the decoder decrypts four bytes at a time, resulting to 32 payload reads. On the other extreme, shellcodes produced by the

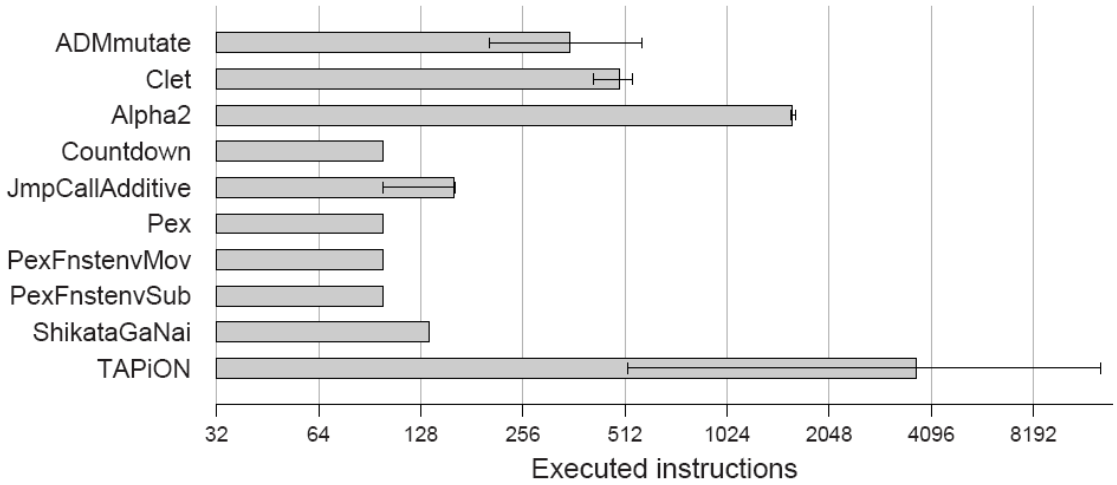

Figure 1 Average number of executed instructions for the complete decryption of the payload

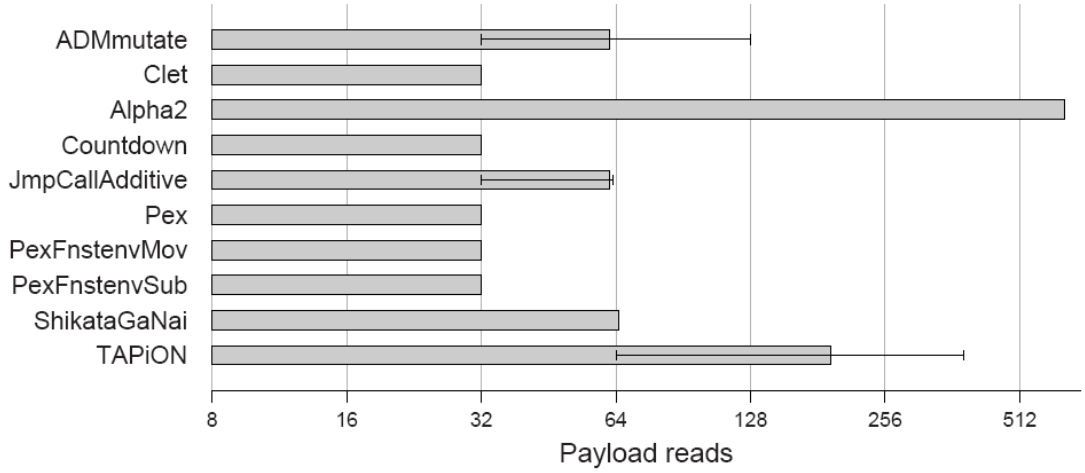

Figure 2 Average number of payload reads for the complete decryption of the payload.

Alpha2 engine perform more that 500 payload reads. Alpha2 produces alphanumeric shellcode using a considerably smaller subset of the IA-32 instruction set, which forces it to execute much more instructions in order to achieve the same goals. Given that 128 bytes is a rather small size for a functional payload, these results can be used to derive an indicative upper bound for PRT = 32. Combined with the results of the previous section, this allows for a range of possible values for PRT from 7 to 31. For our experiments we choose for PRT the median value of 19, which allows for even more increased resilience to false positives.

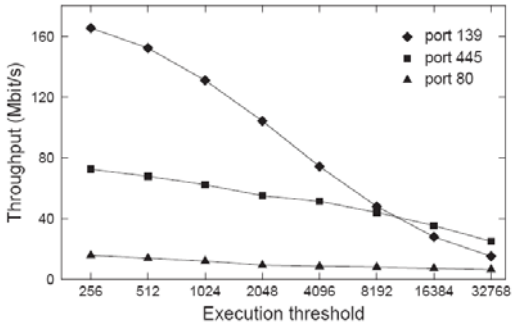

Figure 3 Processing speed for different execution thresholds

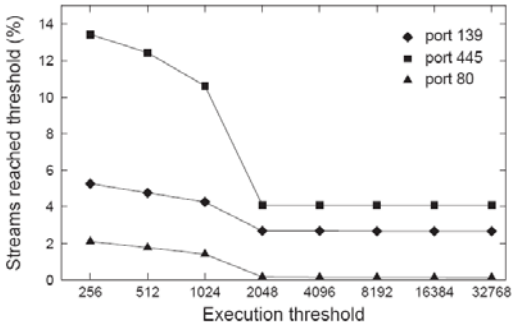

Figure 4 Percent of streams that reach the execution threshold

Detection Effectiveness. To test the efficacy of our detection method, we launched a series of remote code-injection attacks using the Metasploit Framework [17] against an unpatched Windows XP host running Apache v1.3.22. Attacks were launched from a Linux host using Metasploit's exploits for the following vulnerabilities: Apache win32 chunked encoding [18], Microsoft RPC DCOM MS03-026 [19], Microsoft LSASS MS04-011 [20]. The detector was running on a third host that passively monitored the incoming traffic of the victim host. For the payload we used the win32 reverse shellcode, encrypted with different engines. We tested all combinations of the three exploits with the engines presented in the previous section. All attacks were detected successfully, with zero false negatives.

e. Processing Cost

In this section we evaluate the raw processing speed of our prototype implementation using the network traces presented in Table 1. Although emulation is a CPU-intensive operation, our aim is to show that it is feasible to apply it for network-level polymorphic attack detection. One of the main factors that affects the processing speed of the emulator is the execution threshold XT beyond which an execution chain stops. The larger the XT, the more the processing time spent on streams with long execution chains. As shown in Figure 3, as XT increases, the throughput decreases, especially for ports 139 and 445. The reason for the linear decrease of the throughput for these ports is that some streams have very long execution chains that always reach the XT, even when it is set to large values. As XT increases, the emulator spends even more cycles on these chains, which decreases the overall throughput. We further explore this effect in Figure 4, which shows the percent of streams with an execution chain that reaches a given execution threshold. As XT increases, the number of streams that reach it decreases. This effect occurs only for low XT values due to large code blocks with no branch instructions that are executed linearly. For example, the execution of blocks that have more than 256 but less than 512 valid instructions, reaches a threshold of 256, but completes with a threshold of 512. However, the occurrence probability of such blocks is reversely proportional to their length, due to the illegal or privileged instructions that

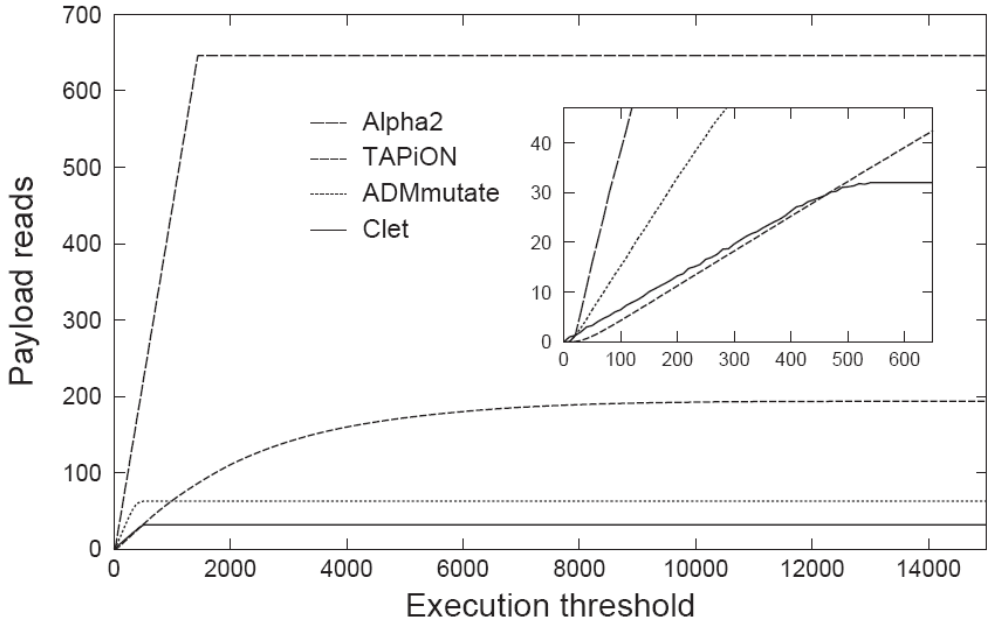

Figure 5 The average number of payload reads of Fig. 2 that a given execution threshold allows to be executed

accidentally occur in random code. Thus, the percent of streams that reach XT stabilizes beyond the value of 2048. After this value, XT is reached solely due to execution chains with endless loops, which usually require a prohibitive number of instructions in order to complete. In contrast, port 80 traffic behaves differently because the ASCII data that dominate in web requests produce mainly forward jumps, making the occurrence of endless loops extremely rare. Therefore, beyond an XT of 2048, the percent of streams with an execution chain that stops due to the execution threshold is negligible, reaching 0.12%. However, since ASCII web requests do not contain any null bytes, the zero-delimited chunks optimization does not reduce the number of execution chains per stream, which results to a lower processing speed. Figures 3 and 4 represent two conflicting tradeoffs related to the execution threshold. Presumably, the higher the processing speed, the better, which leads towards lower XT values. On the other hand, it is desirable to have as few streams with execution chains that reach the XT as possible, i.e., higher XT values that increase the visibility of endless loop attacks. Based on the second requirement, XT values higher than 2048 do not offer any improvement to the percent of streams that reach it, which stabilizes at 2.65% for port 139 and 4.08% for port 445. At the same time, an XT of 2048 allows for a quite decent processing speed, especially when taking into account that live incoming traffic will usually have relatively lower volume than the monitored link's bandwidth, especially if the protected services are not related to file uploads. We should also stress that our prototype is highly unoptimized. For instance, a threaded code [21] emulator combined with optimizations such as lazy condition code evaluation [22] would result to better performance. A final issue that we should take into account is to ensure that the execution threshold allows

polymorphic shellcodes to perform enough payload reads to reach the payload reads threshold and be successfully detected. The complete decryption of some shellcodes requires the execution of even more than 10000 instructions, which is much higher than an XT as low as 2048. However, as shown in Figure 5, even lower XT values, which give better throughput for binary traffic, allow for the execution of more than enough payload reads. For example, in all cases, the chosen PRT value of 19 is reached by executing only 300 instructions.

3. Address Space Randomization

Sophisticated worms that use precomputed hitlists of vulnerable targets are especially hard to contain, since they are harder to detect, and spread at rates where even automated defenses may not be able to react in a timely fashion. The objective of Network Address Space Randomization (NASR) is to harden networks specifically against hitlist worms. The idea behind NASR is that hitlist information could be rendered stale if nodes are forced to frequently change their IP addresses. There are two proposed approaches for the deployment of NASR: a DHCP-based one and the Transparent Address Obfuscation (TAO) box. The DHCP-based implementation, although it imposes the lowest administration overheads, may induce passive failures on hosts that change their addresses when connections are still in progress. The risk of such collateral damage also makes it harder to perform address changes at the timescales necessary for containing fast hitlist generators. Rather than controlling address changes through a DHCP server, using TAO, network elements transparently change the *external* address of internal hosts, while ensuring that existing connections on previously used addresses are preserved without any adverse consequences.

a. DHCP-based implementation

A basic form of NASR can be implemented by configuring the DHCP server to expire DHCP leases at intervals suitable for effective randomization. The DHCP server would normally allow a host to renew the lease if the host issues a request before the lease expires. Thus, forcing addresses changes even when a host requests to renew the lease before it expires requires some minor modifications to the DHCP server. Fortunately, it does not require any modifications to the protocol or the client. We have implemented an advanced NASR-enabled DHCP server, called Wuke-DHCP, based on the ISC open-source DHCP implementation[1]. To minimize the “collateral damage” caused by address changes we introduce two modules in our DHCP implementation: an activity monitoring module, and a service fingerprinting module. The activity monitoring module keeps track of open connections for each host with the goal of avoiding address changes for hosts whose services could be disrupted. In our prototype, we only consider long-lived TCP connections (that could be, for example, FTP downloads). More complicated policies can be implemented but are outside the scope of our proof-of-concept implementation. Wuke-DHCP communicates with a flow monitor that records all active sessions of all hosts in the subnet. The flow monitor responds with the number of active connections that are sensitive to address changes.

Service fingerprinting examines traffic on the network and attempts to identify what services are running on each host. The purpose of service fingerprinting is two-fold. First, we want to supplement activity monitoring with some context to make address change decisions by indicating whether a connection failure is tolerable by the end-system. Second, we want to avoid assigning an address to a host that has significant

overlap in services (and potential vulnerabilities) with hosts that recently used the same address. For instance, randomization between hosts with different operating systems, e.g., between a Windows and a Linux platform appears as a reasonable strategy. Our implementation of service fingerprinting is rudimentary: we only use port number information obtained through passive monitoring to identify OS and application characteristics. For instance, a TCP connection to port 80 suggests that the host is running a Web server, and port 445 is an indication that a host might be a Windows platform. In an operational setting, more elaborate techniques would be necessary, such as the passive techniques described in [2, 3] and active probing techniques implemented as part of open-source tools[4, 5, 6, 7].

In our implementation, we use three timers on the DHCP server for controlling host addresses. The refresh timer determines the duration of the lease communicated to the client. The client is forced to query the server when the timer expires. The server may or may not decide to renew the lease using the same address. The soft-change timer is used internally by the server to specify the interval between address changes, assuming that the flow monitor does not report any activity for the host. A third, hard-change timer is used to specify the maximum time that a host is allowed to keep the same address. If this timer expires, the host is forced to change address, despite the damage that may be caused.

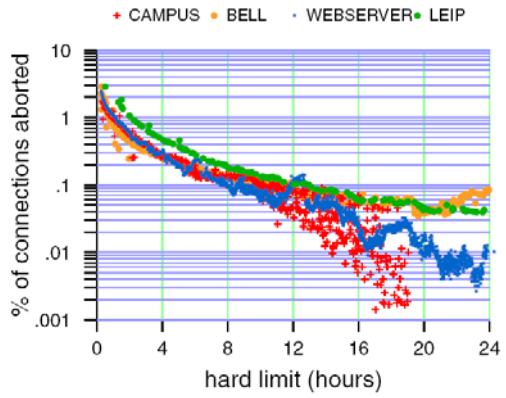

Figure 6 Percentage of aborted connections as a function of the hard change limit

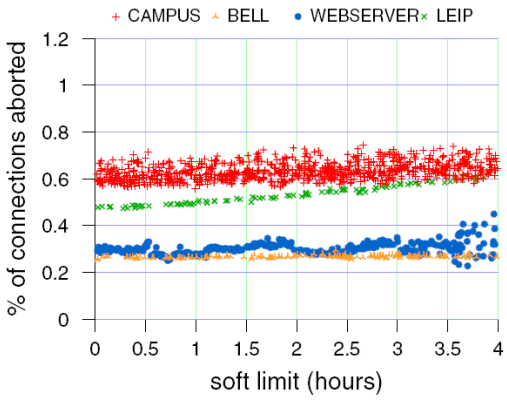

Figure 7 Percentage of aborted connections as a function of the soft change limit

b. Transparent Address Obfuscation

The major drawback of the DHCP-based implementation of NASR is the damage caused in terms of aborted connections. The damage depends on how frequently the address changes occur, whether hosts have active connections that are terminated and whether the applications can recover from the transient connectivity problems caused by an

address change. Although the results suggest that the failure rates are small when measured in comparison to the total number of unaffected connections, the failures may cause significant disruption to specific services that users value a lot more than other connections, such as long-lived remote terminal session (e.g., ssh), etc. Furthermore, the acceptable operating range of DHCP-based NASR does not fully cover the likely spectrum of hitlist generation strategies. In particular, there are likely scenarios that involve very fast, distributed hitlist generation, which cannot be thwarted without extremely aggressive address changes. Aggressive address changes in the DHCP-based NASR implementation have a profound effect on connection failure rates, and the approach hereby becomes less attractive. The damage depends on how frequently the address changes occur, whether hosts have active connections that are terminated and whether the applications can recover from the transient connectivity problems caused by an address change. As shown in Figures 6 and 7, the damage varies from 0.01 to 5%. Experiments were done using traces collected at different network environments: a one-week contiguous IP header trace collected at Bell Labs research[8], a 5-day trace from the University of Leipzig[9], a 1-day trace from a local University Campus, and a 20-day trace from a link serving a single Web server at the institute of the authors. However, as we need to perform randomization in small timescales, where the failure rates wave between 3 and 5%, failure rates may not be acceptable. We can avoid network failures by using *Transparent Address Obfuscation*, an approach which needs more deployment resources than the standard NASR implementation.

The idea behind the mechanism is the existence of an “address randomization box”, called from now on “TAO box”, inside the LAN environment. This box performs the randomization on behalf of the end hosts, without the need of any modifications to the DHCP behavior. TAO box controls all traffic passing by the subnet(s) it is responsible for, analogous to the firewall or NAT concept. The address used for communication between the host and the box remains the same. We should note that there is no need for private addresses, unlike the case of NAT, as end hosts can obtain any address from the organization they belong. The public address of the end host – that is the IP that outside world sees – changes periodically according to soft and hard timers. Old connections continue to operate over the old address, the one that host had before the change, until they are terminated. The TAO box is responsible for two things. First, to prevent new connections on the old addresses (before randomization) reaching the host. Second, to perform address translation to the packets based on which connection they belong, similar to the NAT case. Until all old connections are terminated, a host would require multiple addresses to be allocated.

An example of how the TAO box works is illustrated in Figure 8. The box is responsible for address randomization on the 11.22.70.0/24 subnet, that is it can pick up addresses only from this subnet. Initially the host has the IP address 11.22.70.40 and TAO box sets the public IP address of this host to 11.22.70.50. The host starts a new SSH connection to Host A and sends packets with its own IP address (11.22.70.40). The box translates the source IP address and replaces it with the public one, setting it to 11.22.70.50.

Simultaneously, the box keeps state that the connection from port 2000 to Host A on port 22 belongs to the host with behind-the-box address 11.22.70.40 and public address 11.22.70.50. Thus, on the Host A side we see packets coming from 11.22.70.50.When Host A responds back to 11.22.70.50, box has to perform the reverse translation. Consulting its state, it sees that this connection was initiated by host 11.22.70.40 so it rewrites the destination IP address. After an interval, the public address of host 11.22.70.40 changes. TAO box now sets its public address to 11.22.70.60. Any connections initiated by external hosts can reach the host through this new public IP address. As it can be seen in Figure 3 the new connection to Host B website has the new public IP as source. Note that in the behind-the-box and public address mapping table host now has two entries, with the top being chosen for new connections. The only connection permitted to communicate with the host at 11.22.70.50 address is the SSH connection from Host A. For each incoming packet, the box checks its state to find an entry. If no entry is found, then packet is not forwarded to the internal hosts, else the "src IP" field of the state is used to forward the packet. As long as the SSH connection lasts, the 11.22.70.50 IP will be bound to the particular host and cannot be assigned to any other internal host. When SSH session finishes, the address will be released. For stateless transport protocols, like UDP or ICMP, only the latest mapping between public and behind-the-box IP address is used.

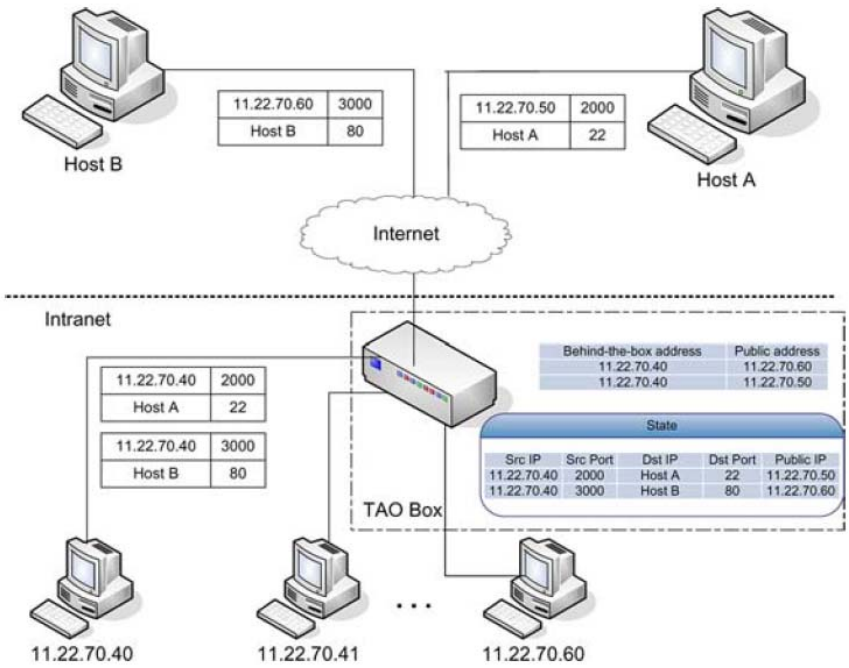

Figure 8 An advanced example of NASR using the TAO box. Host has two public IP addresses, one (11.22.70.50) devoted for the SSH session to Host A and the other (11.22.70.60) for new connections, such as a HTTP connection to Host B.

c. Simulation study

The drawback of the TAO box is the extra address space required for keeping alive old connections. An excessive requirement of address space would empty the address pool, making the box abort connections. We tried to quantify the amount of extra space needed by simulating the TAO box on top of four traffic traces. The first two traces, CAMPUS and CAMPUS(2), come from a local university campus and include traffic from 760 and 1675 hosts respectively. All hosts of this trace belong to a /16 subnet. The second trace, BELL, is a one-week contiguous IP header trace collected at Bell Labs research with 395 hosts located in a /16 subnet. Finally, the WEBSERVER trace is a 20-day trace from a link serving a single Web server at our institute. In this trace, we have only one host and we assume it is the only host in a /24 subnet. In our simulation, the soft timer had a constant value of 90 seconds, while the hard timer varied from 15 minutes to 24 hours.

The results of the simulation are presented in Figure 9. In almost all cases, we need 1% more address space in order to keep alive the old connections. We measured the number of hosts that are alive in several subnets. We used full TCP scans to identify the number of hosts that were alive in 5 subnets: our local institute, a local University campus and three subnets of a local ISP. Our results indicate that 95% of the subnets are less than half-loaded and thus we can safely assume that this 1% of extra space is not an obstacle in the operation of the TAO box. However, the little extra address space needed derives from the fact that subnets are lightly loaded. For example, the 760 hosts of the CAMPUS trace correspond to the 1.15% of the /16 address space. In Figure 10, the relative results of the previous simulation are shown. On average, 10% more address space for hard timer over one hour is needed, which seems a reasonable overhead. In the case of the WEBSERVER trace the percentage is 100% but this is expected as we have only one host.

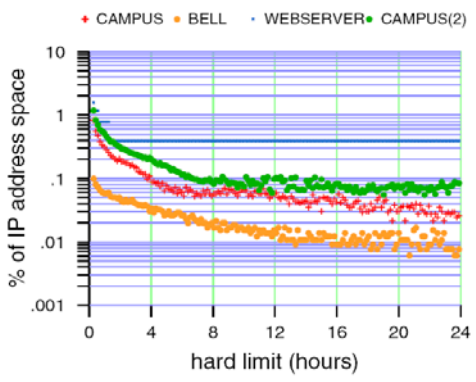

Figure 9 The percentage of extra IP space needed

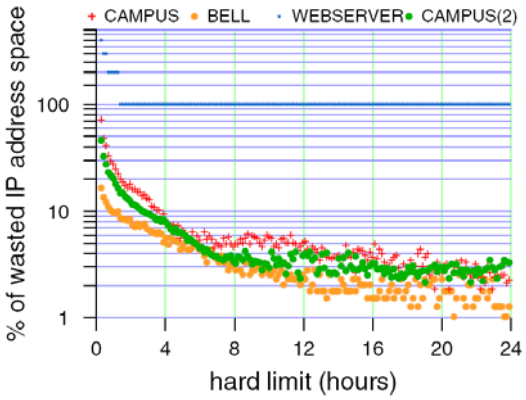

Figure 10 The percentage of extra IP space needed relative to the load of subnets

4. Conclusions

In this document we presented the implementation and performance details of our two approaches that enhance network security: emulation-based polymorphic attack detection and network address space randomization.

Traditional defense techniques rely on static analysis are insufficient, because they can be bypassed using techniques such as simple self-modifications. We explore the feasibility of performing more accurate analysis through network level execution of potential shellcodes by employing a fully-blown processor emulator on the NIDS side. We have examined the execution profiles of a large number of shellcodes produced using various generators and identified properties that can distinguish polymorphic shellcodes from normal traffic with reasonable accuracy. Our analysis indicates that our approach can detect all known classes of polymorphic shellcodes, including those that employ certain forms of self-modifications that are not detected by previous proposals. Furthermore, our experiments suggest that the cost of our approach is modest.

Network address space randomization has shown that hitlist worms can be significantly slowed down and exposed to detection if hosts are forced to change their address frequently enough to make the hitlists stale. However, the implications of changing addresses in a DHCP-based implementation hamper the adoption of this defense, as it can cause disruption under normal operation and cannot be performed fast enough to contain advanced hitlist generation strategies. Transparent Address Obfuscation (TAO), offers more leeway for administrators to more frequently change addresses, while at the same time eliminating the problem of “collateral damage” in terms of failed connections, when compared to the DHCP-based implementation. The experiments presented in this deliverable demonstrated that the cost of TAO in terms of additional address space utilization is modest and that the operation of the system is transparent and straightforward.

5. References

[1] Internet Systems Consortium Inc. *Dynamic host configuration protocol (DHCP) reference implementation*. http://www.isc.org/sw/dhcp/.
[2] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy. *Transport layer identification of P2P traffic.* In IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, pages 121–134, New York, NY, USA, 2004. ACM Press.
[3] S. Sen, O. Spatscheck, and D. Wang. *Accurate, scalable in-network identification of P2P traffic using application signatures.* In WWW '04: Proceedings of the 13th international conference on World Wide Web, pages 512–521, New York, NY, USA, 2004. ACM Press.
[4] *THC-Amap*. http://thc.org/releases.php, 2004.
[5] *DISCO: The Passive IP Discovery Tool*. http://www.altmode.com/disco/, 2004.
[6] *Fingerprinting: The complete documentation*.
http://www.l0t3k.org/security/docs/fingerprinting/, 2004.
[7] *Fingerprinting: The complete toolsbox*.
http://www.l0t3k.org/security/tools/fingerprinting/, 2004.
[8] NLANR-PMA Traffic Archive: Bell Labs-I trace.
http://pma.nlanr.net/Traces/Traces/long/bell/1, 2002.
[9] NLANR-PMA Traffic Archive: Leipzig-I trace.
http://pma.nlanr.net/Traces/Traces/long/leip/1, 2002.
[10] S. McCanne, C. Leres, and V. Jacobson, "*Libpcap*," 2006, http://www.tcpdump.org/.
[11] R. Wojtczuk, "*Libnids*," 2006, http://libnids.sourceforge.net/.
[12] jt, "*Libdasm*," 2006, http://www.klake.org/~jt/misc/libdasm-1.4.tar.gz.
[13] P. Bania, "*TAPiON*," 2005, http://pb.specialised.info/all/tapion/.
[14] K2, "*ADMmutate*," 2001, http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.
[15] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "*Polymorphic shellcode engine using spectrum analysis*," *Phrack*, vol. 11, no. 61, Aug. 2003.
[16] B.-J. Wever, "*Alpha 2*," 2004, http://www.edup.tudelft.nl/»bjwever/src/alpha2.c.
[17] "*Metasploit Project*," 2006, http://www.metasploit.com/.
[18] "*Apache Chunked Encoding Overflow*," 2002, http://www.osvdb.org/838.
[19] "*Microsoft Windows RPC DCOM Interface Overflow*," 2003,
http://www.osvdb.org/2100.
[20] "*Microsoft Windows LSASS Remote Overflow,*" 2004, http://www.osvdb.org/5248.
[21] J. R. Bell, "*Threaded code*," *Comm. of the ACM*, vol. 16, no. 6, pp. 370–372, 1973.
[22] F. Bellard, "*QEMU, a Fast and Portable Dynamic Translator*," in Proceedings of the USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.