ABSTRACT
With fast worms and polymorphic shellcode becoming a reality of today’s internet, questions are being raised about the usefulness of some network-based detection techniques. This paper provides a brief survey of current network-based detectors of worms and shellcode attacks, with a focus on their resilience to current and future attacks. We conclude that network-based techniques may not be appropriate for detecting such attacks in the future.

Keywords
Worm, Polymorphism, Shellcode, Network-based Intrusion Detection, Content-based Filtering, Anomaly Detection, Signature

1. INTRODUCTION
Ever since fast-spreading network worms have appeared on the scene they have been one of the tougher problems in today’s network security world. The problem with these worms is obvious: they can saturate the population of vulnerable hosts too quickly for human intervention and give an attacker the immensely powerful ability to execute arbitrary code on the victim hosts. How to deal with this problem has been the subject of many many papers, and a final word on the subject is not in sight. In fact, the developments around network worms seem to have turned into a “arms race” reminiscent of the developments in computer viruses. [3, 9]

These worms work by exploiting vulnerabilities in server software that allow an attacker to execute arbitrary code. When seen from this perspective, they are just a self-replicating form of a type of attack that has existed for a long time. For lack of a better term we will refer to these attacks as “shellcode attacks” in this paper, after the name given to the code an attacker uses to establish control over the victim host.

The solutions which have been proposed for these kinds of attacks so far generally fall into two categories: host-based detection and network-based detection. Host-based approaches work by instrumenting the software and/or hardware of the host, which usually carries some performance penalty. The required modification of existing hardware/software has made this kind of solution harder to implement and deploy commercially.

Network-based solutions work by scanning the network traffic to the host for malicious or anomalous traffic and are generally designed for performance in order to cope with the large amount of traffic which passes over a link. These solutions often build upon the existing developments in network intrusion-detection devices.

An interesting development in network-level detection of worms is that authors increasingly acknowledge that their techniques are vulnerable to circumvention by a determined attacker. If the goal is 100% accuracy, the verdict seems to be that network-level detection is not up to the challenge if future threats are taken in consideration. [14, 5]

This does not mean that network-level solutions are no longer relevant, of course. The question we will try to answer in this paper is:

What role can network-based techniques play in defending against worms and shellcode attacks in the future?

To answer this question we will consider the following subquestions:

- What is the current state-of-the-art of network-based shellcode and worm detection?
- How resilient are the various discussed detection methods against current and future attacks?

The first subquestion will be answered by a literature study. To answer the second question we will consider the opinions of the authors of the papers we have already discussed in answering the first subquestion. We will also consider the opinions of various other papers on the subject written from both the angles of defense and attack.

In section 2.1 we will explain the methodology used in finding the sources for this paper. In 2.2 we will introduce in some more detail the types of attacks that we will consider. Section 3 contains an overview of current developments in network-based shellcode detection, grouped by the type of technique employed. Section 4 considers the utility of the discussed techniques in various current and future attack scenarios, while paying attention to how these techniques cooperate with host-based defensive techniques. Section 5 contains our conclusions and presents some recommendations for future work.

2. CONTEXT OF THE STUDY
2.1 Literature search and selection
We used various ways to gather the papers for our research. Most of the papers were found and retrieved through the ACM’s online database but we also used the services CiteSeer and IEEE Xplore. Our initial keywords were “shellcode”, “detection”, “network based”, “worm”, “polymorphic”, “signature”, “anomaly”, “IDS” in various permutations.
Since the aim of this paper is to find out how network-based detection techniques can help protect hosts against shellcode or worm attacks we disregarded a rather large amount of research which deals with detecting existing infections on large networks and gateway hosts.

In our initial selection we focused mostly on getting a broad sample of techniques. We then proceeded to consider the list of papers which cite the found papers as well as the papers listed in the bibliography, repeating until we felt confident that most of the important papers had been found.

We supplemented this selection by searching for conferences covering our topic, both by searching the internet and scanning bibliographies for papers published in proceedings. We searched the papers submitted to a few relevant conferences manually or using a search engine where available.

Since this covers quite a broad field we were forced to make a further selection in the papers we could use. We generally preferred newer papers and papers with a higher number of citations. We also paid attention to the purpose of our paper and gave precedence to papers which included extensive discussions of possible attacks against their techniques.

The detection techniques considered in this paper are:

- Newsome et al. “Polygraph” (Signature generator)
- Wang et al. “Anagram” (Anomaly detection)
- Zanero et al. “Unsupervised learning techniques for an intrusion detection system” (Anomaly detection)
- Polychronakis et al. “Network-Level Polymorphic Shellcode Detection Using Emulation”. (Emulator)
- Wang et al. “SigFree” (Static analysis)
- Li et al. “Hamsa” (Signature generator)
- Cavallaro et al. “Lisbeth” (Signature generator)
- Christodorescu et al. “malware transformer”. (Static analysis)
- Udo Payer et al. “Execution chain evaluation”. (Emulation)
- Toth et al. “Abstract Payload Execution”. (Static analysis)

We had some trouble finding recent literature on network-based anomaly detection systems. The search terms “anomaly”, “content based”, and “IDS” were used (amongst others) but not much turned up.

2.2 Defining shellcode attacks

The goal is to review the detection of shellcode attacks, which we here take to mean any attack where the attacker manages to execute native code of his choice. This includes many kinds of worm attacks.

A shellcode was originally code that provided the attacker with access to a shell (command interpreter) on the victim host, thereby providing the attacker with a convenient way to control the host. In order to reach this goal the attacker first needs to somehow get his shellcode into the memory of the victim host. Then he needs to exploit a vulnerability in a server process on the victim host to transfer control to his shellcode.

Once control is transferred to the shellcode it can use the services provided by the operating system with the privileges of the process being exploited.

2.2.1 Exploit and control flow

There are many different kinds of vulnerabilities that can be exploited to allow execution of arbitrary code. Providing a taxonomy of such vulnerabilities is beyond the scope of this paper, but we will provide a brief overview of the constraints they place on the attacker.

Most techniques to transfer control to code of an attacker’s choice involve overwriting a memory location that contains the memory address of a piece of code that will be executed at a later time.

By overwriting this location with the address of data supplied by the attacker, control flow will be passed to this data instead of to the intended piece of code. One disadvantage of this scenario is that the attacker generally has to be able to put his shellcode into a known location in memory for an attack to work, since this location must usually be encoded into the exploit. Since modern programs often allocate most of their memory dynamically this may not be easy to achieve. [11, 16]

There are two common ways to overcome this problem. One way is to increase the chances of success by prepending a large amount of NOP (no-operation, or “do nothing”) instructions to the actual code. In that case one only has to make sure the chosen memory address falls somewhere in the NOP area, and the processor will execute all the following NOP instructions, leading right into the real code.

A second possibility makes use of existing code in the process being exploited to transfer control to the data supplied by the attacker. The most simple form is the “register spring”, where the location of the attacker’s data is still located in one of the machine’s registers when control is seized. The attacker then only has to find an address that contains the bytes that make up the machine instructions that transfer control flow to the address contained in this register.

This latter technique looks much more reliable, but it is sensitive to very small differences in the binary representation of the program being exploited. These differences may be introduced simply by compiling the software on a different system. This means it is often not possible to create a generic attack using this technique for open-source software, which is usually compiled on many different systems into slightly different binary executable files. In contrast, commercial software is often shipped as a binary file which will be the same on every system (until a new version comes out). [8]

2.2.2 Decoder and shellcode

As mentioned, the code to execute must be stored in memory by the victim process before control flow is diverted. In many cases the process being exploited will do some checks on the data it receives or perform some operations on it before storing the receive data in memory. This can be a problem for the construction of shellcode, since the code needs to be valid machine instructions once it is stored in memory.
For this reason the attacker may choose to add another step to the attack. For example, if the process being exploited accepts only uppercase letters, the attacker will use a short piece of code that consists of instructions whose binary representation contains only ASCII uppercase letters to decode his actual shellcode and then pass control to it.

A decoding step can also be used to evade intrusion detection systems. By generating a random decoder and encoding the shellcode accordingly the same shellcode will have a completely different binary representation.

Since the data a decoder works on usually directly follows the decoder, and the location of this data is not known exactly when the decoder starts to execute, many decoders need to first obtain the address of the currently running code. Such code is known as “GetPC” code, since it retrieves the value of the processor’s Program Counter, which contains the address of the currently executing instruction. Because there is a limited amount of ways to implement such code and it is part of the decoder stage GetPC code is one of the more obvious parts of shellcode for which signatures could be created.

The actual operation of the shellcode itself is not very relevant to this paper, so it will not be discussed.

3. OVERVIEW OF NETWORK-BASED DETECTION TECHNIQUES

Let us start by identifying some of the recent work in the area of network-based detection techniques. The method used to search for and select the discussed projects is described in section 2.1.

We have grouped the found techniques into three categories. The only motivation behind the chosen categorization was to keep projects which use similar concepts and techniques together.

The first two categories correspond to the two well known approaches to intrusion detection: “signature-based techniques” attempt to detect known malicious traffic, while “anomaly-based techniques” attempt to identify anomalous traffic based on knowledge of normal traffic patterns.

The third category contains projects which are related by their use of knowledge about the native instruction set of the target host.

3.1 Signature-based techniques

Also known as misuse-based techniques, these detection methods work by scanning traffic for evidence of known attacks. While such systems started out as simple string searchers, the current literature is a lot more complex. To start with, a modern “signature” can take many different forms. And while signatures used to be made by hand, modern techniques often deal with fast-spreading worms by developing ways to automatically generate signatures from suspicious content.

3.1.1 Types of signatures

What does a signature look for? There must be some characteristic that is unique to a certain attack for a signature to be created.

Some techniques [10] operate on the assumption that even with modern polymorphic attacks (see 4.1.1) there is a part of an exploit that must stay invariant, because it is essential to putting the victim program in a the proper state for exploitation. The intuitive explanation is that bugs tend to appear in little-used functionality and edge cases, which require uncommon input to trigger.

Another possible target of signatures is the exploit code itself. Though this is often dismissed as a dead end because of advances in polymorphic encoders (see 4.1.1), some researchers have reported success with these techniques against the current crop of polymorphic encoding engines.

Two other systems are Lisbeth [2] and its predecessor Hamsa [7]. These systems use a multiset of tokens for a signature, which specifies a number of substrings each of which must occur a given number of times in order for the signature to match. Signatures are automatically generated.

3.1.2 Signature generation

In order to generate signatures for a worm one first has to have samples of the worm. It is not always obvious how such a set of samples would be obtained. One option is by generating the worm samples by hand using publicly available sourcecode for worms and polymorphic encoders [7]. This does not really help with protection against new worm outbreaks, however.

Another method uses an external flow classifier that marks a sequence of network traffic as suspicious or normal. Such a classifier generally consists of some kind of network or host-based anomaly detection system. In this case it is obvious that the quality of the flow classifier will have a significant impact on the quality of the generated signatures.

PAYL [18] uses a n-gram based anomaly detection system as a flow classifier, and correlates ingoing to outgoing traffic to reduce false positives on the assumption that worms produce outgoing traffic which is substantially similar to the data that caused the initial infection. Unfortunately, some legitimate protocols also exhibit such behavior (like NTP and various Peer-To-Peer protocols).

3.2 Anomaly-based techniques

Anomaly-based techniques detect attacks by generating an alert when unusual input is encountered. In other words, instead of modeling possible attacks the normal traffic is modeled. The two interesting variables here are what kind of modeling is being done and where the training data comes from.

3.2.1 Training data

In order to create a model of normal data, we first need a sample of normal data. Such a dataset can be created by manually siting attacks from a dataset, but this is obviously a lot of work. To make matters worse, in many cases the model has to be recreated for each deployment since the definition of normal traffic changes from site to site. [18] This makes it a very attractive idea to use the traffic both as training data and as test data. Unfortunately, this enables a whole new set of attacks on the training component of a system (see 4.1.2).

A system which uses a number of well-known unsupervised learning systems can be found in Zanero et al [19]. Unfortunately, they do not provide a comprehensive overview of their attack resilience so this will not be considered further.

3.2.2 Modeling
A simple way of modeling traffic is by calculating a n-gram frequency distribution. This involves sliding a n-byte window over the data and counting the frequency of each n-byte sequence. In the trivial case, this reduces to a byte value frequency count. For higher-order n-grams the space requirements go up quickly. Anagram [17] solves this by not calculating frequencies but only a seen/not seen value, and storing these values in space-efficient Bloom filters.

3.3 Emulation and static analysis

There are a number of network-based techniques that attempt to detect the presence of malicious executable code in network traffic. This can be done either by the assumption that any executable code of a certain length is likely to be malicious, or by attempting to detect certain behavioral patterns commonly seen in shellcode.

The most prevalent instruction set architecture for servers and personal computers at the moment is Intel’s x86 architecture. This architecture has two properties that complicate the kind of techniques described in this section: the length of instructions is not fixed, and the majority of random byte combinations decode to valid instructions (it is a dense instruction set). This means that many different instruction sequences can be obtained from the same data, depending on the precise position where one begins the decoding process.

These properties combined with the fact that code may contain branches or loops often make these kinds of techniques computationally expensive. For this reason some authors suggest using a preliminary filtering step before applying these techniques.

3.3.1 Finding the entry point

The entry point refers to the point at which the executable part of the attack traffic begins. As mentioned, finding the position of this point is a difficult problem, especially without knowledge about the vulnerability or program being exploited.

Exploits that use a NOP area may be detected by looking for a sequence of bytes that is executable from every word-aligned offset (see 2.2). Considering the purpose of a NOP area is to carry the flow of execution into the shellcode itself, execution (or simulation, or disassembly) may start at any point in the NOP area [12]. Recall, however, that on the windows platform NOP areas are not commonly used (see 2.2).

3.3.2 Tracing and detection

Tracing of the flow of execution is further complicated by the fact that due to the lack of information about the destination host’s state there may be uncertainty about which path execution would take in some cases. One solution is to simply follow every possible path [12], though this may be computationally expensive.

Once one or more possible execution paths are identified a decision needs to be made whether the obtained instruction sequence constitutes a threat. There are many ways to do this.

The most simple way is to count the maximum number of instructions executed [16], which assumes that no intentionally executable data is ever present in normal traffic. While this assumption may hold to a certain degree for mostly-text protocols it is insufficient for binary protocols due to the denseness of the x86 instruction set.

Alternatively, one of the possible flows of execution may be selected and passed to an external classifier [4, 12]. The idea is that by passing an execution trace instead of raw binary data to a classifier, the problem of polymorphic encoders and self-modifying code can be circumvented. [4]

One interesting approach is generating a byte spectrum from the executed instructions and classifying this spectrum using neural networks [12].

Other techniques detect behavior which is common to many polymorphic encoders, such as writing to an area of memory which is then executed [3].

4. RESILIENCE OF CURRENT TECHNIQUES

In this section we evaluate the projects and general techniques described in the previous section with regard to their resilience against attack. We identify two types of attacks. The first category consists of targeted attacks, which are for the most part specific to a certain kind of detection technique. The second category are generic attacks, which work against multiple techniques.

4.1 Resilience against targeted attacks

Resilience to targeted attacks is important not only because hackers may specifically target the detection technique that their target network is using. If a certain detection technique becomes popular enough, even attackers not focused on a specific victim will certainly attempt to craft their attacks to bypass that technique.

Since we are discussing attacks which target a specific group of detection techniques, we will structure our discussion in the same way as section 3. We will discuss a number of attacks against each of the categories of detection techniques from section 3 and proceed to give a short evaluation of each category.

4.1.1 Signature-based techniques

Recall that signature-based techniques work by detecting known malicious patterns. Thus, attacks against signature-based detection techniques aim to make the attack traffic diverse enough that a network-based system fails to notice the similarity between two different occurrences of the same attack.

4.1.1.1 Polymorphism

Polymorphism is the property that two instances of the attack’s executable code have a completely different binary representation. Polymorphism is described in more detail in 4.2. We discuss here only the specific implications for signature-based systems.

It is obvious that by definition a properly implemented polymorphism should defeat signature matching on the code part of an exploit. Polygraph is still able to find signatures in the code generated by various current polymorphism engines, but the author concedes that this is probably due to implementation deficiencies in the polymorphism engines. [10] It should also be remarked that this paper generates signatures for exploits in HTTP and other mostly-text traffic, in which binary exploit strings will be fairly noticeable in any case.
4.1.1.2 Further obfuscation
An attack cannot consist merely of executable code, and these other parts may still be matched by a signature.

Systems such as Polygraph depend on the existence of invariant bytes\[10\], byte sequences which are essential to the function of the exploit. In order to get good specificity these bytes must not occur in normal traffic. The analysis provided by \[5\] indicates that such necessary invariants do not exist in sufficient quantity in a selection of recent worms. This would seem to suggest that any success signature-based detection systems currently have in catching these worms is perhaps due to the laziness of the worm authors.

One area which is often difficult to disguise is the literal value that is used to overwrite a pointer to hijack control flow (see 2.2). \(^2\) A binary string such as a pointer is very noticeable in a mostly-text protocol like HTTP. Not every vulnerability needs such a pointer to exploit it successfully, but the vulnerabilities exploited in many recent works do.

There are only a few ways to (partially) disguise this part of an exploit. When a NOP area is used the low bits of this value may be changed slightly without much consequence, but this is often not enough to fool detectors. When a register spring technique is used one could select the register spring location from a list of known usable locations, but the list of such locations that remains usable on many different versions of software is usually rather small.

4.1.1.3 Training attacks
When signatures are automatically generated the signature generator may be vulnerable to malicious training data attacks. See 4.1.2.2 for more details on such attacks.

Recent systems such as Lisbeth provide reasonable defense against training attacks, though they sacrifice some speed and false positives to do so.

4.1.1.4 Evaluation
At this time hand-crafted signatures are still a very useful tool for protecting networks. Similarly, systems that identify polymorphic worms automatically might work on past and current worms. Especially for binary protocols, however, it is likely that exploits will appear for which no signature can be made with an acceptably low level of false positives.

4.1.2 Anomaly-based techniques
Anomaly-based techniques work by detecting unusual data in network traffic. This means that attack traffic can only pass undetected if it explicitly looks “normal” to a certain detection algorithm. This presents a major challenge to the attacker.

4.1.2.1 Mimicry attacks
As mentioned, the obvious way to evade detection by an anomaly detection system is to craft attack traffic to look normal. This is known as a mimicry attack in most literature, though the term “polymorphic blending attack” is often used when considered from the viewpoint of the attacker. \[6\]

Polymorphic encoders that can mimic a 1-gram model (see 3.2) have been demonstrated already \[6, 15\]. Some papers \[18\] assert that their models are very site-specific and thus too difficult to approximate for an external attacker. While this may not be convincing for 1-gram models, later systems such as Anagram \[17\] do a credible job of hardening the model against outside deduction by using higher-order n-grams and randomizing the parts of traffic that are modeled.

4.1.2.2 Malicious training data
Since what constitutes “normal” traffic may be different between hosts, a new model must be created for each deployment. Since training generally requires a sizable amount of data supervised training is often not an option. Thus, we require some automated way of sorting the input traffic into normal and malicious traffic so that the model can be trained.

4.1.2.3 Dealing with false positives
False positives are a problem with any anomaly detection system, since almost any host is likely to receive a small amount of legitimate traffic that is substantially different from the traffic that host commonly sees. \[17, 19\]

4.1.2.4 Evaluation
All but the most trivial \[18\] anomaly detection systems are remarkably hard to circumvent \[17\], especially when using multiple kinds of detection.

Unfortunately, these systems tend to produce false positives in such numbers that automated responses are not possible and expensive human monitoring may be needed. However, when coupled with another second detection system that is still very accurate but has less false positives network-based anomaly detection systems may still be the way of the future.

Some host-based systems claim such characteristics, and indeed some authors suggest using a network-based anomaly detection system together with host-based instrumentation. \[17\] Others have concluded that such a system is the most effective defense against worms. \[1\]

4.1.3 Emulation and static analysis
Emulation techniques are attractive since they provide a tool against the powerful evasion method of polymorphism. However, there is a large amount of anti-emulation tricks from the fields of computer viruses and software protection tools.

4.1.3.1 Host state dependence
Network-based emulation techniques rely on the observation that exploit code tends to depend on the state of the victim process as little as possible, in order to be robust against changes between different version of the victim software and different host operating systems. \[13\] (See also section 2.2)

By breaking this assumption in any but the most trivial way the attacker can prevent emulation from succeeding. We would like to note that doing this in such a way as not to exclude any vulnerable hosts from infection might take some effort on the part of the attacker.

4.1.3.2 Self-modifying code

\(^2\)Indeed, many of the example signatures shown in the various discussed papers contain the sequence 0xFF 0xBF 0xBF, which are the high bytes of a memory address on the stack on Linux systems.
Static analysis techniques [12, 16] are easily defeated by self-modifying code, a well-known technique from computer viruses. [13]

Because static analysis techniques do not actually execute the instructions being analysed they will fail to notice when one instruction modifies the following instructions. This can (for example) lead the analysis program to conclude the code is invalid when it is in fact fully functional shellcode.

4.1.3.3 Removing characteristics of shellcode

Since traffic may sometimes contain legitimate executable code (file up/downloads, automatic updates) and because random data sometimes contains a significant amount of executable instructions there needs to be some way to detect shellcodes specifically. Emulation-based techniques might look for GetPC code, NOP areas, or self-modifying behavior, but none of these elements is strictly required for a successful exploit. [13]

4.1.3.4 Execution time

An acknowledged problem with emulation techniques is that it may take a while before the shellcode exhibits any interesting behavior. The simulation cannot be continued indefinitely since this would take too much time. Random byte sequences do sometimes produce an infinite loop, and it is hard for a detector to differentiate between a random infinite loop and a purposeful busy loop that is meant to disrupt analysis [12, 13].

4.1.3.5 Evaluation

Though Christodorescu et al. achieved good results with “unpacking” polymorphic encoders to make shellcodes more vulnerable to classic signature matching, network-based emulation techniques are fundamentally limited by their lack of knowledge about the host — though some techniques do their best to cope with this deficiency [13].

The resilience of these techniques to evasion is not completely clear, but the considered papers already identify some weaknesses that do not appear to be immediately solveable. An example is prepping a random-generated loop to the shellcode to trick the emulator into believing it has entered an infinite loop and aborting the emulation.

Because of their doubtful resilience against evasion and combined with the relatively high computational cost we do not see these techniques as having much of a future, except in their host-based variants.

4.2 Generic anti-detection techniques

The techniques discussed in this section are not aimed at one specific detection technique, but are general techniques used to obscure the contents of an attack. We discuss them here to help clarify the limits of network-based shellcode detection. Due to lack of data we will not discuss the resilience of the techniques identified in section 3 to these attacks in much detail.

4.2.1 Polymorphism

Polymorphism is the family of techniques by which an instruction sequence is rewritten to provide the same functionality with a completely different instruction sequence. This is done to make the code more difficult to detect. The technique was originally developed by the makers of computer viruses, but is now being used in shellcodes. [15]

A properly implemented polymorphism engine makes the code itself impossible to detect for signature-matching algorithms. Polymorphism engines have also been developed that try to generate the new instruction sequence so that it matches the model of “normal” data as used by anomaly detection systems. [15]

4.2.2 Application-specific encodings

Many of the more complex network-facing applications of today support various methods by which the same input can be given to the application in different ways.

- Escape sequences. For example, in URLs %41 is the same as the letter A.
- Different encodings. For example, UTF-16, UTF-8 or plain ASCII.
- Encryption. Some network applications may support streams encrypted with a choice of encryption algorithms such as 3DES, AES and Blowfish.

Any such functionality provides more ways to vary the representation of an attack. To be able to successfully defend against such techniques the network-based detector will probably have to know how to do some application-specific decoding. Allowing it to scan encrypted streams might even require modifications to the hosts that are to be protected. [5]

4.2.3 IDS avoidance techniques

There are a number of ways attackers can try to make sure only the victim host sees their attack code, and not any IDS system which may be listening. Most of these techniques focus on tricking the host and the IDS system into reassembling the network packets into two different flows. This can be done by exploiting different timeout settings, different TTL distances from the attacker, different rules in handling overlapping sequence numbers and other such properties of the network stack which may differ between the target host and the IDS system.

Proper reassembly of packet flows is usually seen as a problem that is orthogonal to the detection of malicious data in those flows, and for this reason we will not go into more detail here. We mention it here because a complete network-based system must naturally still provide a solution for this problem.

5. CONCLUSIONS

We have identified a number of recent developments in the field of network-based detection systems. We found examples of both anomaly detection systems and misuse detection systems that claim to be effective against attacks such as polymorphism. The developers of these techniques are currently well aware of the kinds of methods attackers can use to evade their techniques and their modern papers are accordingly much more resilient against evasion. The effectiveness of signature matching techniques against sophisticated attackers is quite limited, but the prevalence of simple attacks will ensure their continued relevance for the foreseeable future.

An interesting category of detection techniques was identified that uses knowledge of the instruction set of the target host to identify the executable part of shellcode. These techniques appear to be reasonably effective against current polymorphic techniques. Although they provide a novel means of detecting shell-
code attacks, they are computationally expensive and their resilience against evasion techniques is doubtful.

In evaluating the studied defense techniques we concluded that none of the network-based techniques offer perfect protection against shellcode attacks. This is to be expected, since a network-based technique does not possess full information on the host environment. However, this does lead us to conclude that a network-based technique alone cannot prevent a targeted attack or an advanced fast worm.

Existing and proposed systems mostly work against the current generation of worms, but these are certainly circumventable by a determined attacker. Thus the choice is between using network-based systems as a filter for more processing-intensive host based techniques, or trusting in the writers of worms and shellcodes to be lazy enough to be caught by purely network-based defenses.

The latter choice may not be as bad an option as it appears, since intrusion detection systems are still not widely deployed and attackers may simply choose to target only unprotected hosts. If the network to be protected is likely to be a specific target for hackers the outlook for network-based techniques is bleak, however.

Network-based anomaly detection systems detect nearly all kinds of attacks, but their high false positives are a disadvantage. This might be solved by using an anomaly detection system as trigger for a more resource-intensive host-based system. Network based anomaly detection systems have more difficulties with binary protocols, and encrypted data may mean that it is not able to work at all.

Network-based emulation systems are mostly aimed at solving the problem of the polymorphic encoder. Since they do not have access to the host context, however, they offer no guarantees about accuracy and may be easy to circumvent. More research is necessary to determine how big a problem these techniques pose to attackers.

In order to successfully defend against future attacks some sort of host-based detection seems inevitable. This reduces network-level techniques to a role as only the first in di
tering layers of protection to be much higher degree than we have seen so far. Making an effective detection system is getting harder, and more attention needs to be paid to the integration with other layers of protection to be sure that a network-based technique is actually adding value.

As our final recommendation, we suggest that future research into the area of network-based detection techniques should justify the appropriateness of using a network-based approach to a much higher degree than we have seen so far. Making an effective detection system is getting harder, and more attention needs to be paid to the integration with other layers of protection to be sure that a network-based technique is actually adding value.

REFERENCES


