Global Information Assurance Certification Paper

Tyson Kopczynski
GCIH Practical Assignment Version 1.5c

Mutated Code

Introduction:

As part of the escalating arms race between hackers and security engineers, a new weapon designed to evade Intrusion Detection Systems (IDS) has recently made its way into the public arena. In March 2001 during the CanSecWest conference K2 made public his shellcode mutation engine called ADMutate **(1)**. ADMutate is an API that is designed to change around the code structure of buffer overflow exploits. This polymorphism of the code structure is done in order to mask the signatures of the attack from IDSs by giving the hacker the ability to create variants on the fly. When used properly, this technique can render the signature analysis portion of any IDS useless. Handing the developers of IDSs a setback in their current arsenal to combat network intrusions. However, it is unclear if this is really a major setback for those developers. The general consensus from both the hacking and security communities is one of mixed feelings. On one side, there is the belief that this program is the start of a new period where hackers will gain the advantage over the security community. Others express very little concern over the technique because it attacks a portion of IDSs that is both out of date, and backed up by other means of detecting an intrusion **(1)(2)**.

Exploit Details

Name	ADMutate; ADMMutate
Variants	None Yet
Operating System	Multi-Platform
Protocols/Services	Tries to evade signature analysis programs in an IDS system.
Brief Description	Polymorphic shell-code toolkit and libraries for IDS evasion.

A background to IDS:

Network security is comprised of confidentiality, availability, and integrity. With the advent of the Internet the need for security systems has increased dramatically. In order to ensure aspects of a secure network it becomes essential for a network administrator to protect his or her network from attacks. For a short period of time it was popularly thought that firewalls would provide all the protection needed to limit the reach of most attacks. However, a multitude of techniques soon emerged to bypass the barriers firewalls put in place. Now lacking a definitive way to

protect their networks and besieged under attacks, the security community reacted by maturing IDS technology to handle the process of discovering and analyzing attacks on their networks.

The concept of Intrusion Detectors has been around since the 60's. However, it wasn't until the late 80's and early 90's that advances in the technology brought together the basic elements found in all of today's IDSs: the analysis engine and event generator. These basic elements have since then been expanded on, and today most IDSs now resemble the structure found within the Common Intrusion Detection Framework (CIDF[1]) model **(3)**:

- E-Boxes: Event Generators
  - Termed as the sensory organ of IDSs because it provides information about events to the rest of the system.
- A-Boxes: Analysis Engines
  - Analyzes the streams of data from the event generators.
- D-Boxes: Storage Mechanisms
  - Defines the means and structure by which the huge amount of data that is produced from the E-Boxes and A-Boxes is stored.
- C-Boxes: Countermeasures
  - Can be a simple alarm system to actual countermeasures that allow the IDS to take action and prevent further attacks (I.E. shutting down TCP connections, modifying router filter lists).

To analyze the system or network data for possible attacks, IDS developers most commonly use the following two techniques: misuse detection and anomaly detection. Misuse detection analyzes system or network activity looking for patterns that are defined based on attack signatures. This reliance on signatures is why misuse detection is more commonly called signature analysis. The other form of activity analysis is anomaly detection. Anomaly detection is the analysis based on determining abnormal patterns of activity in a system or network. Anomaly detectors do this by assuming that normal activity will appear to be different than activity associated with attacks **(11)**.

Signature and anomaly analysis are the most common means used in IDSs for detecting attacks. However, they are not the only means. Other means might include **(9)**:

[1] By using this CIDF model as an example of the layout of IDSs, my goal here was to try and illustrate the basic components within IDSs. However, this actually is a standardization/interoperability effort that is very theoretical. So the layout that is given in this paper should not be considered as the industry standard when dealing with naming conventions and direct layout.

- Frequency or threshold crossing
- Correlation of lesser events
- Honeypots and Protocol Analysis[2]

Currently there exist two types of IDSs: the Network based (NIDS) and the Host based (HIDS).

- HIDSs deal with the security and integrity of a single host. The HIDSs role is to determine if the host is in the process of being attacked or if an attack has already resulted. HIDSs usually do this by monitoring event logs, processes, and host-based traffic in either real or near real time and use either signature or anomaly analysis to identify attacks. Some current HIDS products are Tripwire, Real Secure OS Sensor, and Dragon Squire.
- NIDSs deal with the security and integrity of the network. To do this a NIDS might passively capture all the packets that are traversing a network using a sniffer. At this point, an NIDS might either be using signature or anomaly detection to analyze the data for attacks. Some current NIDS products are Snort, Real Secure ISS, and Dragon.

Signature analysis and NIDSs:

In most IDSs signature analysis currently is the most popular method used for detecting attacks. With NIDSs signature analysis is mostly based on how attacks are performed on TCP/IP and the fact that attack signatures can be developed characterizing those attacks. To detect a possible attack the signatures are compared with the information gathered from a sniffer. Then based on if the attack signature matches the sniffer information, a certain event is then triggered.

Here is an example of a simple signature analysis. If a NIDS that is protecting web servers has a string entry for “phf” as a possible cgi attack and if the sub-string entry “phf” was discovered in a stream of data, it would then most likely generate an event, alarm, or attack responses **(3).**

Sample of a Snort Signature (4):
alert tcp !$HOME_NET any -> $HOME_NET 21 (msg:"OVERFLOW-FTP-x86linux-adm";flags:PA; content:"|31 c0 31 db 17 cd 80 31 c0 b0 17 cd 80|";)

Excerpt from ADMwuftp.c (wu-ftp 2.42 exploit):
“\x31\xc0\x31\xdb\x17\xcd\x80\x31\xc0\xb0\x17\xcd\x80”

A possible problem:

[2] Honeypots and Protocol Analysis will be discussed later in this paper.

Several years ago, signature analysis was a very powerful tool. However, it has always been subject to a very serious problem. The signature is only a hard coded entry in a database. This database needs continuous updating by the developers of IDSs in order to protect networks and systems from the growing variants of attacks on them. With this in mind, K2 uses a method called polymorphism the coder's way of forcing code to exist in several distinct forms at once as a means for bypassing NIDSs. This technique in coding has been around for many years. In fact viri developers heavily use polymorphic algorithms in their code. In most cases they will use polymorphism in conjunction with encryption to hide the main body of the virus using a polymorphic decryptor **(5)**. The main body of the virus is encrypted, while the polymorphic decryption routine is kept in plaintext. This is a very attractive technique to viri developers because the time needed to write a polymorphic virus is significantly reduced because the polymorphic decryptor is small and only performs one task **(6)**. In most cases, these types of viruses are not generally considered truly polymorphic, since the actual body of the code does not vary, only the decryptor cipher is really polymorphic in nature.

With true polymorphic viruses the code is placed through an algorithm or function that produces unique code while keeping it equivalently the same as the old code. Because of this, the method is very popular to use in producing a variant for code quickly. In a properly designed engine this process of mutation should hold true until the n^{th} time without any change to functionality when the code is passed through it. A simple example of this is, if a function is looking for output in the form of an integer regardless what that integer is, then the engine would most likely just be a random number generator that produces a new and different integer at runtime **(2)(4)**.

The algorithm or function that generates the polymorphic code is called a mutation engine. These engines can be used while compiling the code or integrated within the code body itself. There are several different effects or uses that can be derived from the use of mutation engines. One course is to vary the code by any of the following **(5)**:

1. Dispersing the code with "noise" functions (a.k.a NOPs "No Operation Programs, programs designed to load a unused registers with an arbitrary value).

   Example one byte NOP instructions (4):

   - clc – clear carry flag
   - cld – clear direction flag

- stc – set carry flag
- std – set dirction flag

2. Interchanging independent functions.
3. Varying the function sequences (**Subtract A from A, with Move 0 to A**).

Another example is when the engine comes in the form of an object module. In this form, calls are added to the assembler source code that links the engine and random-number generator modules together **(5)**. This is the approach that ADMutate resembles the most. ADMutate is in fact an API that can be called by your exploit (some shellcode) at runtime to mutate its behavior. The effect is that every time the exploit is run its operating signature will be different each time. So any NIDS that only looks for the signatures of attacks would easily be bypassed **(2)(4)**.

How the exploit works:

How is all this done? As K2 points out in his ReadMe for ADMutate **(1)**, most polymorphism techniques can easily be translated into an exploit coder. This is because, in general, most buffer overflows have a large number of NOPs that can be easily mutated to perform other tasks without changing any of their main functionality. For ADMutate he gives the following example in the form [NNNN][SSSS][RRRR] with 700 bytes for NOPs “N”, 80 bytes for shellcode “S”, and 200 bytes return address “R” (1KB buffer size).

1. The shellcode is encoded with an (xor) function that not only makes it unique from the original code but also acts like a cipher to encrypt the code. This is done using the apply_key function, which is using a (2x16bit) keyspace that can generate over 100,000 possible keys.
2. Using the apply_jnops function all of the NOPs are substituted with instructions that are completely different, however, equivalent in length.
3. Now that the second step is complete the decoder gets randomly generated with the apply_engine function. This function mutates the decoder so its operational signature is different every time it is generated.

Techniques used for this:

- Randomly Generated Instructions
    a. Loading a register with some data the possible instructions can be used.
        i. PUSH DATA
        ii. POP REGISTER
        iii. MOVE DATA, REGISTER
        iv. CLEAR REGISTER
        v. ADD DATA, REGISTER
    b. Now randomly replace each instruction of the decoder.
- NOP Padded Instructions
    a. During decoder generation spare CPU registers are left open.
    b. In the open registers other NOPs are plugged in.
- Out-of-Order Decoder Generation
    a. Order of the instructions are offset **(4)**,

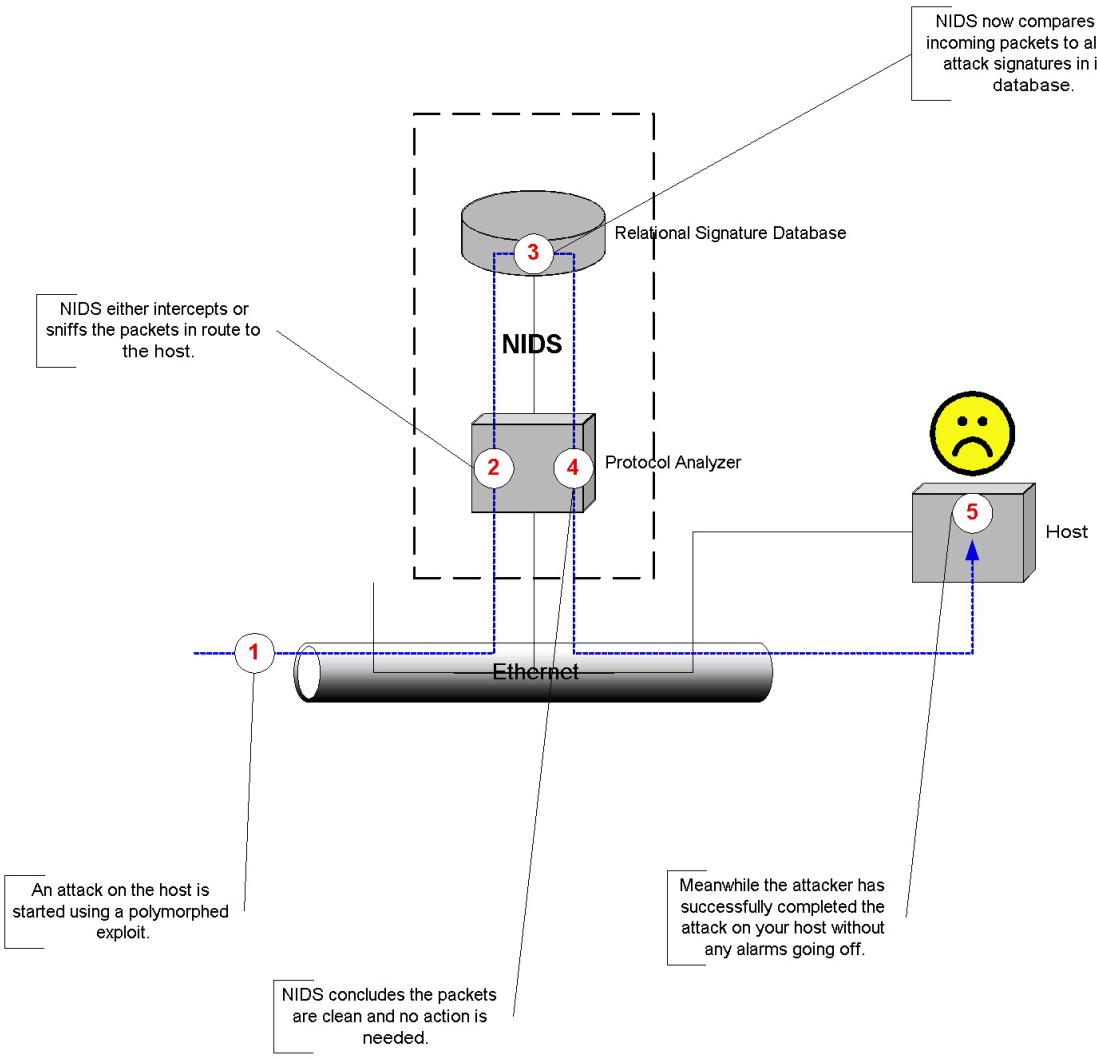
```
\x31\xc0          xorl    %eax,%eax
\x88\x46\x07      movb    %eax,0x7,(%esi)
\x89\x46\x0c      movl    %eax,0xc(%esi)
\xb0\x0b”         movb    $0xb,%al
```

becomes

```
\x31\xc0          xorl    %eax,%eax
\xb0\x0b”         movb    $0xb,%al
\x88\x46\x07      movb    %eax,0x7,(%esi)
\x89\x46\x0c      movl    %eax,0xc(%esi)
```

- Multiple Code Paths

4. The shellcode at runtime will now have a different operational signature each time it is run.
5. The last thing that K2 contends with is any shellcode restrictions based on the OS that the code is running on when dealing with the tolower(), and toupper() functions. Shellcode restrictions can be found in the mut.banned in the struct morphctl mut control structure.



*****The following information is taken directly from K2's ADMutate readme with some minor changes to try and make if more readable for everyone.*****

Files	Description
ADMmutapi.h	API glue
ADMmuteng.[ch]	ADMutation API
m7	Demo of the API
vulnerable	Local vulnerable program
vulnerable-remote	Remote vulnerable program (run from inetd)
exp	Exploit that should exploit "vulnerables"
expx	Demo's the API in a typical exploit
qp.c	Demo remote qpop linux exploit
zdec.*	Skeleton's for the decoders
Functions	
API Functions	init_mutate, apply_key, apply_jnops, apply_engine
Optional Function	apply_offset_mod
Control Structure	struct morphctl mut
mut.arch	Set to one of (DISABLE, IA32, SPARC, HPPA, MIPS)
mut.lower	Enables tolower() resilient code
mut.upper	Enables toupper() resilient code
mut.omodulate	Modulate the offset
mut.cipher	Cipher that is used
mut.cipher2	Second cipher that is used
mut.banned	Char string excluded from final shellcode
Function Format	
init_mutate(struct morphctl *)	Initializes pointers to OS specific structures, functions, and limits all specified in mut.arch.
apply_key(buff,E,N,srtuct morphctl *)	Takes input buff, substitutes E bytes of it at location buff+N with encoded bytes. Also finds a key that works with processing options. Call it after init_mutate.
apply_jnops(buff,N,struct morphctl)	Takes buff and substitutes the first N bytes with some "junk nops."
apply_engine(buff,E,N,struct morphctl)	Takes buff and generates a decode engine then places it at buff+N-strlen of the engine.
apply_offset_mod	Takes buff, mods it N amount at location buff+X while adhering to options set in morphctl. Return will then be unique on execution.

Sample code you would add into exploit (IA32 Version)

```
#include "ADMmutapi.h"
struct morphctl *mctlp;
struct morphclt mut;
mut.upper = 0; mut.lower = 0; mut.banned = 0; mctlp = &mut; mut.arch = IA32

init_mutate(mctlp);
apply_key(buff, strlen(shellcode), nop-1, mctlp);
apply_jnops(buff, nop-1, mut);
apply_engine(buff, strlen(shellcode), nops-1, mut);
```

How to protect against it:

Update, Update, Update! It cannot be stressed enough about keeping all of your systems up to date with current patches. ADMutate is for use with exploit coders. This usually means that the exploits that are mutated by this API will most likely be for well-known exploits. So if systems are kept on the most current security patches then it might be possible to protect your systems even if tools like ADMutate bypass your NIDSs.

In general when deploying a security solution, it should be complemented with well-rounded security policies and practices. Here is a small checklist of some things that should be considered when dealing with NIDSs.

Network Considerations:

1. Harden and secure all intrusion detectors. Software based NIDSs sit on top of server OS's, and are subject to the same attacks as all your other servers.
2. Have a policy in place to update your signatures on a regular basis.
3. Review these policies on a regular basis.
4. Deploy your NIDSs either on each network segment or on segment boundaries.
5. Deploy HIDSs on either all your systems or at least on mission critical systems. HIDSs will be able to alert you to possible attacks on the system if your NIDSs happen to be by-passed.
6. Install other Network Security Devices on your network. I.E. Firewalls, and Proxies.
7. Develop an understanding of network traffic that is considered trusted. As part of that list determine the origin and destination of the traffic.
8. Now apply ACLs through out your network based on this understanding of traffic traveling in and out of your network. The goal here is to only allow required traffic in and out of your network. I.E. Do I really need to be passing icmp throughout my network?
9. Review your network traffic needs on a regular basis and update the ACLs on a regular basis. I.E. After removing all the webservers from the network we don't need to be

passing http inbound anymore.

10. Routinely run vulnerability analysis tools on your network to determine possible vulnerabilities your network.

11. Correct the problems found by the vulnerability analysis tools.

NIDS Purchasing Considerations:

1. Simple. Employ the use of an NIDS that does not simply rely on signature analysis. Remembering that when you deploy your NIDS you will have to complement it with other network security tools, devices, and a sound security policies and practices.

In recent years the security community has made significant progress in expanding or refining the techniques that NIDSs use to detect attacks. Some of the more exciting advances are:

Honeypots
Honeypots are systems that are designed to resemble something that an intruder can attack. A honeypot might consist of a dummy machine that only serves a function to log all the attacks performed on it or specialized software that resembles a full network for intruders to attack while logging all their activities. Honeypots can serve as an early-alarm or hostile-intent assessment system **(10)**. Examples of Honeypots: NAI CyberCop String, Specter, Netcat.

Padded Cells
Although somewhat different then a honeypot, a padded cell works in parallel with an IDS acting as a special holding cell for attackers. When the NIDS detects an attacker it transfers the attacker to the padded cell host where the attacker is contained within a simulated environment. Like honeypots the simulated environment might be filled with things like systems that the intruder can attack, all the while being monitored by the NIDS **(11)**.

Protocol Analysis
Protocol analysis is Often confused with signature analysis because half of the intrusions detected using protocol analysis are pattern based, while the other half are not. However, protocol analysis is not just a database of signatures. What protocol analysis does is break the packet payload into different fields and then check each field one-by-one looking for any anomalies. Anomalies are based on the definitions of how a particular application protocol (ICQ, ARP, NetMeeting, ICMP, etc.) are supposed to function **(12)**.

Example: large input
Expected length: 100 bytes
Discovered length: 1000 bytes
Diagnosis: buffer-overflow

Advances in AI for NIDSs

Currently most NIDSs have some form of AI built into to handle such things as statistical anomaly detection. The most exciting possibility is that AI, in theory, could be used to detect light variations in attacks signatures. This provides a possible solution to attacks spawned by ADMutate.

Source code:

The source code be found on K2's website http://www.ktwo.ca/ under his security section. If you are planning on playing around with this tool (API) I strongly suggest reading his readme before continuing.

Description of variants:

Currently there are no **"known"** variants of ADMutate since it was just released to the public a couple of months ago. However, considering most of the techniques that K2 uses in ADMutate where pioneered by viri developers and have been around for years, variants will probably sprout up in the underground pretty quickly. A good thing is that in its current form ADMutate is not very user friendly and requires some level of knowledge in programming to even attempt to use it. Leaving the click and hack, script-kiddies at loss for using this.

Wrap Up:

Just how much of a threat is ADMutate to current NIDSs. Currently there are very mixed feelings on how much of a threat this tool brings to the table. On one spectrum you have NIDS developers claiming that this is not a showstopper at all. The other end is comprised of individuals thinking that ADMutate and future tools like it could pose a high threat to the IDS concept. As in his README K2 explains, "Simple signature analysis unfortunately, cannot provide very high levels of assurances **(2)**." Clearly he is correct when dealing with NIDSs that determine attack intent from a signature compared to database entry. Polymorphic shellcode is something that these systems simply cannot handle. Making the matter more serious is that K2 has found the very popular IDS ISS RealSecure 5 to be vulnerable to ADMutate. He is quick to point out that RealSecure can handle some polymorphic shellcode attacks because it can detect suspicious packets via protocol analysis, however, it can only do this with

protocols that it can understand (FTP/POP/IMAP) vs. (HTTP/RPC/DNS). Further making matters more complicated, ISS has been so bold as to claim that they are immune to all attacks launched by polymorphic shellcode, contradicting claims made by K2 **(8)**.

> *… ISS RealSecure uses different algorithms and methods of detection to determine when a buffer overflow attack happens. These algorithms are not affected by ADMmutate. ISS RealSecure has been confirmed as not vulnerable to the ADMmutate tool.*
> *ISS X-Force is researching adding additional algorithms to identify both specific ADMmutate attacks and generic polymorphic attacks to be provided in conjunction with the buffer overflow alert. Providing this additional information can help identify the sophistication level of an attacker.*
>
> *Conclusion:*
> *ISS RealSecure has been confirmed as not vulnerable to the ADMmutate evasive technique… **(7)***

ADMutate regardless whether it is a high threat or not, is a new way of thinking that very boldly points, once again, more weaknesses in NIDSs. For any NIDS vendor/developer to simply brush this off without looking at the implications of this tool and the techniques that it employs can be very dangerous position. Yes, currently most NIDSs that employ other methods on top of signature analysis to determine hostile intent might have no problem dealing with polymorphic shellcode. However, many of these "other" methods can be just as subject to the same flaw that was exploited by ADMutate. These methods still require, to some extent, that the NIDS vendor come up with some pattern, signature, DAT, software update, etc. to correctly identify and handle new potential attacks. Even such things as protocol analysis fall into this realm because the engine still needs to be updated every time there are changes/additions to the application protocol subset that the NIDS needs to handle. Even the concept of a Honeypot falls short if the logging mechanics can't detect the attacks it's trying to attract.

Maybe with future enhancements underdevelopment for IDSs current evasion techniques will become irrelevant and useless in the near future. However, at this point in time, ADMutate, other tools, and techniques are continually pointing out more and more flaws in IDSs. Eroding the once widely held view that IDSs could be the silver bullet in network security. A position that should have never been taken by security professional to begin with, because, it was the same mistake that was made with firewalls. Simply put the appearance of ADMutate should make clear to the masses that to have a secure network is to be dynamic in the undertaking of securing and protecting that network. There can be no single solution, and the solutions that are employed must constantly be reviewed and upgraded.

References:

1. Lemos, Robert. "New cloaked-code threat to security." 04/02/2001. URL: http://www.zdnet.com/zdnn/stories/news/0,4586,5080532,00.html?chkpt=zdnn_rt_latest (07/18/2001).
2. K2. "ADM ReadMe." URL: http://www.ktwo.ca/c/ADMmutate-0.8.1/README (07/18/2001).
3. Ptacek, Thomas and Newsham, Timothy. "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection." 01/2001. URL: http://www.snort.org/IDSpaper.pdf (07/18/2001).
4. jeru. "Advanced Evasion of IDS buffer overflow detection." URL: http://pr0n.newhackcity.net/~jeru/idsevade.ppt (07/20/2001).
5. Indiana University Knowledge Base. "What are stealth, polymorphic, and armored viruses." 07/07/2000. URL: http://kb.indiana.edu/data/aehs.html (07/21/2001).
6. Yetiser, Tarkan. "Polymorphic Viruses Implementation, Detection, and Protection." 01/23/1993. URL: http://www.bocklabs.wisc.edu/~janda/polymorf.html (07/21/2001).
7. Neohapsis Archives. URL: http://archives.neohapsis.com/archives/sf/ids/2001-q2/0091.html (07/21/2001).
8. Neohapsis Archives. URL: http://archives.neohapsis.com/archives/iss/2001-q2/0018.html (07/21/2001).
9. IIS. "Network- vs. Host-based Intrusion Detection: A Guide to Intrusion Detection Technology." 10/02/1998. URL: http://secinf.net/info/ids/nvh_ids/ (08/06/2001).
10. Graham, Robert. "FAQ: Network Intrusion Detection Systems V 0.8.3." 03/21/2000. URL: http://www.ticm.com/kb/faq/idsfaq.html#11 (08/06/2001).
11. Bace and Mell, "NIST Special Publication on Intrusion Detection Systems." URL: http://www.securityfocus.com/data/library/idsdraft.pdf (08/06/2001).
12. Graham, Robert. "Side Step: IDS evasion vs. protocol-analysis." 03/30/2001 URL: http://www.robertgraham.com/slides/0103cansec/0103cansec.ppt (08/11/2001).