

Antonin Beran
CSCD 340
Final Paper
5/23/07

“Foiling the Cracker”: A Survey of, and Improvements to, Password Security

found at: <http://mack.ittc.ku.edu/cache/papers/cs/5603/http.zSz.zSzwww.ja.netzSzCERTzSzJANET-CERTzSz.zSzKleinzSzD.Klein.Foiling.the.Cracker.pdf/klein90foiling.pdf>

This paper discusses password security both from the technical and social sides. While the author does discuss password encryption algorithms from an algorithm point of view as well as a time point of the view, the major focus of this paper is the human side.

In the early days of Unix, password encryption used a simulation of the M-209 cipher. The problem with this algorithm was that it was too quick. Each encryption only took 1.25 ms, so a cracker could easily go through a 250,000 word dictionary in just a couple of minutes. Facing this problem designers came up with the DES algorithm. This was much slower, on the rate of 280 ms using a faster machine than what was getting 1.25 ms on the earlier cipher. From a technical point this algorithm worked fine for the time. However as one is well aware CPU's have gotten exponentially faster, especially with the consideration of networked computers using divide and conquer techniques to crack the password. In addition to faster hardware, new implementations of the DES encryption algorithm have been developed. This means that once again a cracker can simply brute force a password in a reasonable time. While I am sure there have been new encryption techniques discovered over time, the author doesn't deal with them. Instead he moves on to the main focus of the paper.

The second more serious problem is the social factor, namely users choosing poor passwords that are easily cracked or even just guessed. Although this may come as a surprise to some people in the world, using your user name or something from `/usr/dict/words` is not a good idea if you are interested in a secure password. To prove his point, Klein gathers a list of 15,000 passwords from people all over the world and attempts to crack them. He manages to crack almost 25% of them using simple cracking techniques like common permutations of the user name, their actual name as well as checking common names, `/usr/dict/words` etc. From his research in the matter “word pairs, where a password consists of two short words, separated by a punctuation character.”(4) are a strong class of passwords.

As with any problem there are several possible solutions, all with various trade-offs. One solution that the author discusses is making the password file unreadable except by root. This solution was proposed by Sun Microsystems, but unfortunately doesn't seem to have gained much traction. It is possible that this method has gained more popularity in the years since this paper was published, but since I do not keep up with the Unix world much I cannot say. Another possible solution is running a 'password checker'. This would effectively attempt to crack all the users passwords to whatever degree the administrator decides. While this would at least help to improve password security, there are several problems with it. First off there is a good possibility that the administrator that styles the test will miss some password permutations that are easily cracked, therefore nullifying his efforts. Secondly running this on a system of any size would take an immense amount of time and CPU power, to the point of impracticality. A third option is to make the users change their passwords on a regular basis. However this gives no assurance that the users will be actually changing their passwords to anything secure.

Klein's solution to this problem is a proactive password checker. This would be a customizable program that would immediately check the password the user attempts to create and if the password fails its criteria it would notify the user that their password wasn't strong enough as well as why it wasn't. Not only would this ensure that only strong passwords made it on the system, but it would

educate users on how to create strong passwords.

While most of the paper is fairly common sense to a relatively tech savvy user, it does bring up a couple of interesting points. First off that no matter how advanced encryption technology gets, social engineering to gain passwords is still a major problem without proactive action. For me personally, the concept of word pairs for passwords is something I had not heard of, especially for being as strong as it is. Finally a thought regarding Klein's proactive password checker concept. While I think it is a good idea in theory, I am leery of a program deciding whether my passwords are good enough for it or not. Personally I like the version of the concept I have seen in practice, which is it tells you that it believes your password to be weak, but doesn't actually enforce you changing it.

"Password Security: A Case History"

found at: <ftp.mcc.ac.uk/pub/security/PAPERS/PASSWORD/PWSTUDY.PS>

This article discusses the history of password security algorithms for UNIX time share systems. Specifically it talks about the back and forth battle between the 'good guys' and the 'bad guys' as far as either securing passwords or figuring them out. The focus of the case study is more on the technical side than the social engineering side of password cracking.

In the beginning, time shared UNIX systems used the M-209 cipher machine that was used by the army in WWII. However there were several problems with this encryption scheme. The first was that using a constant key, it was very easy to invert the cipher to gain access to the passwords. This was solved by using the actual password as the key, and a constant as what was encrypted; while this solved one problem, several more still remained. The more serious problem with the M-209 was that if a 'bad guy' was trying to brute force passwords, since it only took 1.25 milliseconds per password check, it was relatively straightforward to run through whole lists to figure out passwords in a short amount of time. To check every permutation of 6 character all lower case words would only take a 107 hours. The article makes a brief mention of how easy it is to guess the majority of user passwords. In a study of 3289 passwords, 86% of them fell into one of several easily guessed categories, rather depressing news all in all.

Several different solutions to the previously mentioned problems have been implemented over the years. First off a new encryption algorithm called DES was created, and it solved one major problem of the older M-209 cipher. Namely it was much much slower, thus making the brute forcing technique much less practical. Secondly there was a movement to make users create passwords that were tougher to crack. If implemented correctly, this can greatly reduce the amount of passwords that a 'bad guy' can simply guess. Lastly, a technique called salting was invented. Basically this is an extension of what the DES encryption algorithm did. By appending a 12 bit quantity to typed password and then encrypting the whole thing, now testing a given character string against a list of encrypted passwords has been multiplied by 2^{12} .

One last fairly small point to discuss is dealing with the security of remote log-ins. The first issue is to make sure not to give any hints if it is the user name or the actual password that is incorrect. Secondly there is an approximate half a second delay when the algorithm is run to check the password. Therefore when the user name is correct, but the password is wrong there is the noticeable delay. An observant cracker could use this to determine if their user name or password is incorrect, thereby reducing the amount of work they have to do. The obvious solution to this problem is to run the algorithm regardless of whether the password is correct or not, thereby eliminating the delay.

This article was a fairly straightforward read about encryption algorithm and the ways to crack them. Ultimately I still believe that no matter how complex the algorithm get, the major security flaw will always be the user, and therefor the 'bad guys' will still continue to use social engineering to gain passwords.

"The Internet Worm Program: An Analysis"

found at:

<http://cobnitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/3092/ftp:zSzzSzftp.cs.purdue.edu:zSzpubzSzreportszSzTR823.pdf/spafford88internet.pdf>

This article is a highly in depth and fascinating read about the first Internet worm. This worm was released on the Internet Nov. 2, 1998, and while it didn't do any real damage it had a profound effect regardless. First off, it did manage to know systems off the Internet and equally importantly it exposed several UNIX security flaws, that although known for some times in some cases, had never been fixed. The majority of the article is composed of a very detailed breakdown of the code present in the worm and how it worked. Most of it is beyond my level of C understanding, but I was able to get the general gist of it regardless. Finally the author analyzes the overall intent of the code as well as trying to explain why the program was constructed and worked how it did.

The two major bugs exploited with the worm involved the 'figurd' program and the sendmail utility. In the case of the 'figurd' program a buffer overflow attack was enough to exploit the program to do what the author desired. Sendmail was exploited by taking advantage of DEBUG code that was left present in the program. Effectively the worm would use the DEBUG command to issue a set of commands to sendmail instead of a user's address. Normally this wasn't allowed, however with DEBUG it was, therefore the flaw.

Here is a very high-level overview of the basic working of the worm. This overview starts at the point at which a machine is about to be infected. First a socket gets established for the vector program to connect to. Secondly the vector program is installed one of two ways a.) With a TCP connection b.) Using a SMTP connection. The vector would then transfer three files; Sun binary version of the worm, a VAX version and the source code of the vector program. Once the files finished transferring the worm would attempt to hide itself by unlinking the binary version of itself, killing its parents and obscuring its argument vector. Now that it was established on the machine it would search the network for other connections and attempt to infect them. The worm utilized three infection strategies; sendmail, fingerd and rsh. Sendmail and fingerd have been already explained so let me now explain the rsh method. All it involved was attempting to open a remote shell and installing the vector program. Finally we get to the meat and potatoes of the program. The worm entered a state machine which consisted of 5 different states. States 1-4 were involved in attempting to crack users passwords on the infected machine. Finally the fifth step simply involved the worm entering an endless cycle of attempting to infect other machines. This is a vastly over-simplified description of how the worm works, but it covers the basic concepts. I don't discuss the password cracking because it mostly involved standard cracking techniques such as using a dictionary, using common permutations of the user name etc.

At the end of the article the author shares several interesting observations about the author of the program. While the sophistication of the networking component of the worm is very high, overall the code is very sloppy and mediocre to say the least. There are multiple examples of unused data, untested calls, incorrect calls and just sloppy code. This makes the author speculate that while the author might have had extensive experience with networking, they were an average C programmer at best. He also speculates that its quite possible that the release of the worm was premature and/or the programmer was simply too lazy to do much if any debugging of it. Personally the way it sounds how some of the code was extremely well written and other parts quite the opposite I think someone managed to scavenge some snippets of code and attempted to mold them with his own creation to create the worm. Unfortunately they didn't have the C knowledge to correctly implement it. It is reminiscent of the script kiddies of today. Either that or the author is correct in his speculation that the author was a very capable network programmer, but had very little experience with different aspects of C programming. Regardless it was interesting to see buffer overflow attacks being used to exploit bugs

in code in the very first web worm when it is still a common attack almost 20 years later. Guess that makes it an 'oldie but goody' eh?

"UNIX Security in a Supercomputing Environment"

found at:

<http://portal.acm.org/citation.cfm?id=76263.76341&coll=&dl=ACM&type=series&idx=76263&part=Proceedings&WantType=Proceedings&title=Conference%20on%20High%20Performance%20Networking%20and%20Computing&CFID=15151515&CFTOKEN=6184618>

This article discusses UNIX system security in the context of a supercomputing environment, although the issues discussed is applicable to all UNIX systems. The author divides security into four different areas: Local User Authentication, Access Control, Integrity, and Least Privilege, Network Privacy and Authentication and lastly Logging and Auditing. For each area he talks about the current systems in place and how they are lacking and how they can be remedied using techniques available on other various 'secure' UNIX configurations.

The area of local user authentication is what most people think of when security is brought up. This subject has been mostly beat to death regarding how to ensure that users don't use easily cracked passwords. While the system can force the user to change their password every so often, or even generate the password for them, both of these implementations have problems. Namely that forcing users to change their password is not guarantee that their new password will be any less easily crackable. If the system generates passwords for the users, there is a high probability that the users will write it down to remember it, leaving it open to be copied or stolen. Also brought up was that the default operation of the password file leaves it completely undefended against it being read, most secure systems employ shadow password files, which hides it. In the end the author recommends a two tier security system which employs password aging combined with a shadow password file and finally some sort of challenge system to make sure that the login attempt isn't from someone who just happens to have the correct user name and password.

The guiding principle of the second area is compartmentalization combined with assigning users the minimum amount of privilege they need to complete the tasks they need to do. The system administrator must be responsible for ensuring that users are assigned the correct amount of permission overall and for individual files. Also the standard UNIX system has the *setuid* and the related *setgid* commands that can change the process privileges to another user. Obviously this has the potential for serious abuse and the recommended solution to this is to eliminate the privilege rights that those programs have if any process rights to them. Regarding compartmentalization the system should use standard control lists and the superuser account should be spread over several accounts.

The last actual security area is controlling network security. As seen by the worm incident in 1988, this is a very serious issue that needs to be addressed. The elephant in the room is the issue of trusted users on remote connections. Since these trusted users have to do through no password checking, if an intruder can gain access to that account, they have a free run of the network; obviously not what the system administrator wants! Interestingly enough this is the location that the author brings up the old M-209 cipher present on UNIX systems compared to the much more robust DES algorithm scheme that 'secure' systems tend to use. Unfortunately he doesn't have any brilliant ideas how to secure a network besides the disabling of trusted accounts, thus forcing password checks as well as ensuring that the encryption algorithm used on the system is DES instead of the older M-209.

Finally he comes to the unpleasant truth that we all know is out there; no system is completely secure. This is when logging and auditing come into play. In the case that the system does get penetrated if there is a functional logging system in place, it can make it much easier to figure out where the problem came from and how to fix it. The standard UNIX logging implementation is woefully inadequate and one should upgrade it to one of the better ones available.

It seems to me that many of the issues the author addresses in this article are still very valid today, which brings up the question of are we actually making any relative progress on the security front? In the age old battle of 'good guys' vs the 'bad guys' it seems that one camp never seems to really ever take the lead for good, they just manage to hold it for a moment. This applies for the strict algorithm side of things regarding such issues as encryption as well as the human side of things such as choosing secure passwords and not letting them get out. I guess only time will tell how the ultimate outcome plays out.

"UNIX Password Security - Ten Years Later"

found at:

http://citeseer.ist.psu.edu/cache/papers/cs/5603/http:zSzzSzwww.ja.netzSzCERTzSzJANET-CERTzSz.zSzFeldmeier_and_KarnzSzcrypto_89.pdf/unix-password-security-ten.pdf

This article is a re-visitation of the article, *'Foiling the Cracker': A Survey of, and Improvements to, Password Security.* In essence it looks at the same password security issues ten years later and sees how the landscape has changed. Unsurprisingly it hasn't changed much.

As one is knows the computing power of computers has steadily reason over the years which in effect makes encryption algorithm that ran slow enough in the past now run too fast. Case in point in the classic DES algorithm that crypt uses. If you combine this fact with speed optimizations possible on the standard crypt implementation the overall increase in speed is scary. Specifically the authors managed to achieve an increase of 102.9 times over the standard crypt implementation. When running this improved crypt on a Sun SPARCStation they could get to a top speed of 1092.8 crypts a second. This is a considerable improvement over past times. One must also keep in mind this is for only one workstation, with parallel processing spread over multiple computers these numbers can get much better.

To improve password security a technique called salting was invented, which basically attempted to make the storage space to store all possible salts too prohibitive. This might have been successful in the days of magnetic tapes, but now technology has passed this by and with the invention of digital cassette tapes storage is no longer a viable concern. This leads to the next issue with password security.

One way to improve the time to brute force passwords is pre-encrypting a large dictionary with possible passwords. With a large dictionary stored on a cassette tape considerable time can be saved. With the author's configuration the cassette tape transfer rate of 250 kb/sec approximately 30,000 trial passwords can be checked a second. Overall the speed when using this technique is 28.6 times faster than simply using real time encryption on a workstation. It is easy to see that the speed increases available to crackers pose a problem to system security.

Just as the initial article did, the authors attempt to show what can be done to improve password security, and their finding are very similar to what Klein came up with 10 years ago. Klein came up with the idea of slowing down the DES algorithm so that brute forcing passwords took too much time to be viable. The authors of the current article come up with several issues with this method. First off not all systems in use are similar in hardware processing power. Therefore what kind of slowed down DES algorithm would work for a CRAY II wouldn't work for an IMB PC. Secondly and more importantly the crypt implementation that runs in the login function is much slower than an optimized crypt that a cracker would use for password breaking. Not only is the cracker implementation much more optimized, but the cracker can afford to dedicate their entire system to the task, while a normal user would hardly want to spend their entire system resources on their login function! One method of password security the article came up with is the use of 'smart' card that would use an electric or infrared link to connect to the system. The 'smart' card would effectively be a basic keyboard. While this would be a secure option, it is hardly viable for remote connection, thus having a major

shortcoming in the increased networked society of today. Lastly they visit the eternal issue of ensuring that user passwords are harder to crack. I won't discuss the methods such as the system assigning arbitrary passwords because they have already been extensively covered in this paper. One new method the authors came up with is using user created phrases as passwords. The passwords would have to be shortened to the required 8 character length for the UNIX system, but if such a method as XOR'ing the words with each other would achieve this while significantly reducing how guessable the password is. One possible issue with this, is that users might rebel against having to enter a lengthy phrase every time they logged into a system. One way around this problem is to use a distributed authentication system. All in all, for any password guessable reduction scheme to work, the users must buy into it and actually use it as intended else it will be all for naught. Considering that systems are still being broken into with easily guessed passwords almost twenty years later, one can guess how successful that endeavor has been.

"A High-Speed Software DES Implementation"

found at:

http://cobnitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/5603/http%3A%2F%2Fwww.ja.net%2FCERT%2FJANET-CERT%2F..%2Ffeldmeier_and_Karnz%2Fdes.pdf/feldmeier89highspeed.pdf

This article examines how the DES algorithm works in extremely excruciating detail. Not only does it cover the basic operation of the detail, but every permutation and optimization possible for every aspect of it. I might be exaggerating here, but if so only by a marginal amount, the article really does cover the bases. In addition the entire article is written in a compact high informative fashion heavy on mathematical formulas. This forces me to be rather high level in my overview of the article otherwise you would be getting a 10 page summary for a 13 page article and neither one of us wants that!

In DES algorithm effectively works as by initially taking a 64 bit password and a 56 bit key. It performed its operations and returns a 64 value. The operations it performs happens in four basic steps. First off it gets the initial permutation from the text. Then it goes through a cycle of 16 rounds swapping the left and right portions of the input in block-transformations. After the 16 rounds of product-transformation/block-transformation it is time for the third step; a overall block transform. Lastly there is a inverse initial block transformation before the DES algorithm returns the 64 bit output.

The meat and potatoes of the encryption happens during the 16 rounds of product/block transformation, so that are bears a closer looking at. The general operation is that the 64 bit block getting passed in gets broken up into two 32 bit blocks (L and R) and the R block gets modified with a transformation using the 48 bit subkey and finally XOR'd with the L block. Ok, so how does this mysterious transformation actually work? Well the initial R block is 32 bits, so first it must get expanded to 48 bits by duplicating some of its bits. Secondly it then gets XOR'd with the 48 bit subkey mentioned earlier. Now the 48 bits gets passed broken up into 6 bit portions that get passed into 8 different 'S boxes', each which modify their input and return a 4 bit value. 8 times 4 is the 32 bit value that the entire transformation returns. Finally that 32 bit value that was returned for the R block is XOR'd with the L block to complete the transformation. This process happens 16 times total for the entire DES algorithm.

This is merely the basic essence of what the DES algorithm does, the remaining portion of the algorithm discusses a myriad of ways for further optimize the speed and space constraints of the algorithm, but none of them fundamentally change the working process. While this paper was relatively interesting in the fact that it really dug into the nitty gritty details of how the algorithm works, there was way too much math and other technical information for my poor tired brain to absorb. I had to take a hour nap just to attempt to vaguely summarize it! However what the article did do very well was give me a really good idea of how complicated encryption algorithm can get while leaving

space for much more in depth analysis of it if I choose to really go over the mathematical formulas it gives as well.

"Cryptographic Protocols over Open Distributed Systems: A Taxonomy of Flaws and related Protocol Analysis Tools"

found at:

kerkis.math.aegean.gr/~dspin/pubs/conf/1997-SafeComp-Formal/html/doc.pdf

This article gives the reader an overview of current cryptographic protocols flaws present in the world of distributed systems as well as a covering of the available cryptographic protocols themselves. The flaws can be broken up into three general categories: functional specification flaws, implementation-dependent flaws, and implementation flaws. As far as the actual cryptographic flaws goes, there are six of them that the authors identify: elementary protocol flaws, password/key guessing flaws, stale message flaws, parallel session flaws, internal protocol flaws, and cryptosystem flaws. According to the paper the definitions of the general protocol flaws are as follows: functional specification - "logical flaw in protocols' high level specification", implementation-dependent - "protocol's specification can result in implementations of which at least one contains the flaw and at least one other does not", and implementation flaws - "faults that occur when a correct specification is incorrectly implemented".

Onto the specific protocol flaws we go. Elementary flaws include such scenarios as the authentication key exchange between two parties. While the general idea behind this is good, the major problem is the key gets encrypted prior to it getting sent, which allows an intruder to insert their own key in the process thus effectively bypassing the security. The ever classic password/key guessing flaw is when the users effectively only utilize a small sample of possible password combinations of the n length permutations possible. This commonly occurs when users choose common words that can be found in a dictionary, or use common variants of those words or people's names etc. There are three sub-categories of this flaw: detectable on-line password guessing attacks, undetectable on-line password guessing attacks and off-line password guessing attacks. The easiest one of these to foil is the first and it can be done simply by keeping track of the number of attempts and the stoppage of service of that account after x attempts fail. However this also can lead to denial of service attacks, so further improvement of techniques to foil this attack need to be implemented as well. A further two security protocols that the authors come up with to assist with security with the first two attacks are: "the authentication server to only respond to fresh requests" and "the authentication server to only deal with requests of verifiable authenticity". As far as off-line passwords attacks go, they simple suggest to make sure that user passwords are stronger using password creation checking software and the like. Stale message flaws occur when the attacker attempts to use genuine message fragments that he shouldn't be able to read or create to penetrate the system. There are both message origin attacks as well as message destination attacks. Parallel session flaws are related to the stale message flaws, but differentiate by the attacker often intercepting communication to fool the system in giving him data that he shouldn't get. These session flaws can be broken up into two main categories; single or multi-role participant. The next protocol flaw covered by this paper are internal protocol flaws. These can happen when one of the participants fail to complete all their necessary actions. Flaws of this type typically are often something like one of the participants failing to check to make sure the message sent was received correctly etc.

There are two main categories of methods used to detect the above mentioned types of protocol flaws. They are 'attack construction tools' and inference construction tools'. To summarize in a nutshell the differences between these two types; attack construction tools main issues is the large amount of possible events they have to check while inference construction tools fail to catch some of the protocol flaws such as parallel session multi-role. Personally for me to really get the most out of

this article I would have to read another one just on the actual tools used to detect the protocol flaws. However it was nice to finally look at additional security flaws that plague the computing world besides simple password cracking.

"The Design and Analysis of Graphical Passwords"

found at:

<http://citeseer.ist.psu.edu>

The article examines how to implement graphical password systems on hand-held devices such as PDA's and if they are more secure than classical textual password systems. The center of the author's argument about graphical passwords being stronger than textual ones is that fact that with graphical systems allows a decoupling between the position of inputs from the temporal order in which the inputs occur. They examine two related but different graphical password schemes, one which uses graphics to assist an underlying textual system and a pure graphical one.

The system the authors came up for a graphical password system augmenting a textual one is relatively simple, but rather effective. It is basically a 'fill in the blanks' system that holds the normal password, but in a cryptographic form such as reverse, shifted to the right once etc. For example the original password might be 'apple' and with the crypto'd version of the word might have been shifted 1 char to the right giving the password of 'eappl'. Basically this system lets the user make their passwords much stronger while only having to remember the cryptographic scheme used to modify it. In addition since the devices these security schemes are small handhelds the actual characters can be displayed on the screen allowing the user an easier time of ensuring they are inputting the correct password. The reasoning for this is that with these small devices they users should have a much easier time to shield the screen from curious onlookers.

The second password scheme the article discusses is the much more interesting one of them and is purely based in the graphical word. Effectively it divides the screen into a grid and lets the user draw whatever design they want on it as the password. What is actually stored as the password is the coordinates ie the 'cells' the drawing went through, the order that they were touched and when and where the pen was lifted up and touched back down. This data of course would have to be stored on the devices with some secure encryption algorithm, but as proven this isn't a problem considering the ones available. The real beauty of this password scheme is that it drastically improves the amount of possible passwords the users will select out of the entire universe of them. Because we are by nature much better at remembering patterns and pictures than simple text users will come up with a much wider range of passwords that they will actually have an easier time to remember than simple text passwords. Not only will there be a larger range of passwords used, but they will be much harder, if not effectively impossible for crackers to educatedly guess at what they might be. There is much less knowledge about any type of common symbols that people associate with than what are common words that people use. This is particularly true when you consider there won't be any of the issues of users using their names or variants of them for their passwords. Overall I think this is a very good scheme, the only drawback is that it is only viable on a small segment of electronic devices ie handhelds with touch screen capabilities. Hopefully in the future when touch screens for computers become common this can be implemented more universally and then I think we will see the difference in security.

"Restricting Network Access to System Daemons under SunOS"

found at:

<http://citeseer.ist.psu.edu>

This article looks at how to secure networks from the vast amount of daemons that run on them

and are necessary for functionality. Specifically this looks at networks using SunOS. At the time of the writing Sun's Network Information Service (NIS) allows any daemon that requests information to get it. This is problematic because this service contains the data that is normally found in /etc/passwd, in other words the encrypted versions of all user passwords. To be more precise NIS merely contains this information and the actual process that allows access to these files is *ypserv*, which is more than happy to give information to anyone that asks. To be fair *ypserv* only gives information to processes with a uid of 0, but it does not do any checking of the originator of the request. This means that any user with any id of 0 can send a process forth to retrieve information and they shall receive it.

The authors bring up three possible solutions for this problem. The first and most commonly used is the firewall. Basically this program determines which packets get let in or out of a 'zone', based on user preferences. While this is quite workable for many network configurations, there are problems with it. First off it makes it more inconvenient for users to pass in and out of the network, due to its security restrictions. Secondly it doesn't work well for FTP's and the like. The second possible solution is called Secure RPC. Basically this protocol works by allowing applications determine what type of authentication they require. However the only two types of authentication for this is UNIX and DES. For secure users DES is highly recommended since UNIX is easily crackable. The last security protocol is what the rest of the article discusses and it involves wrapping programs to enhance security.

Basically this implementation uses the wrapper to check the incoming connection information every time a daemon requests whatever the wrapper is protecting. The wrapper relies on the connection information that the kernel call sends it to be correct so its validation can work correctly. Unfortunately there is no guarantee that the kernel will have the correct information, leaving a possible security hole. Secondly and more importantly on a highly frequently used process there validation of every call simply slows down the system too much to make it viable. So overall while this security implementation does improve security significantly there are still security and performance issues with it that must be resolved before it truly can be considered for widespread implementation.

“Number Theoretic Attacks On Secure Password Schemes”

found at:

<http://citeseer.ist.psu.edu>

This article should be use as cruel and unusual punishment for people. Never before have I seen something that is so completely beyond my comprehension level. Basically this paper discussed some very complex numeric theory regarding password authentication session keys and ways to encrypt them as well as ways to break different schemes. I would attempt to explain the underlying math for some of the schemes, but frankly I have no idea how it works and I would be simply copy and pasting it here without any actual understanding. Instead I will attempt to give you a basic overview of several of the session encryption schemes and their vulnerabilities.

The main scheme discussed here is the Encrypted Key Exchange (EKE) protocol. There are three different variants of it that get talked about; RSA EKE, Diffie-Hellman EKE and ElGamal EKE. First lets examine RSA. A basic description of this algorithm is that the message to be encrypted gets raised to the e th power of n . The *numbers* e and n combine to form the public key. They actually come from permutations of hard to guess prime numbers. However this encryption scheme is vulnerable to a numeric theory attack, and even with several slight changes to it to attempt to make it tougher to crack, it still ends up being vulnerable. Unfortunately no matter how tweaked this algorithm gets, without completely revamping how it works, it remains vulnerable to number theory attacks.

The second main variant of EKE is Diffie-Hellman and it works by using a prime p and a generator g . Both of these values are publicly known. Both sides of the exchange calculate a random number between 0 and $p-1$ and then do some calculations to it to come up with a new value. This value

is also publicly exchanged. Finally both sides raised the last communicated common number by their randomly generated number to form a value that is then mod p . This forms a common key that both sides have. Apparently this is a discrete log problem and it is considered to be extremely difficult to solve with large enough p values. The major flaw with this method is not the actual encryption of the exchange key, but the lack of authentication. If a cracker gains the 'man in the middle' status they can understand all communication between the two sides. It is of course possible to modify Diffie-Hellman to encrypt the communication as well, but that lends itself to different methods of breaks the whole session. However if there is a prior agreed range of values for generating p these security problems can be resolved.

Finally we discuss the ElGamal technique. This method is based on the previously discussed Diffie-Hellman method, but with some significant changes. It still uses p and g , but two numbers get generated and using them a value gets created to get sent to the other party. The other party uses this value along with some complicated calculations to generate a value that allows it to read the message. Unfortunately this algorithm is still vulnerable to the same type of numeric theory attacks that plague the DH-EKE algorithm.

“Detecting Steganographic Content on the Internet”

found at:

<http://citeseer.ist.psu.edu>

There have been claims that terrorists have been using steganography to hide messages on the internet. Steganography is the technique of hiding messages in images. There are three central issues that any steganographic method must consider; capacity, security and robustness. The three steganographic methods that the authors analyze are: JSteg, JPHide and OutGuess. All three of these programs use the same basic method for hiding data inside of images. One very common image format is JPEG and this is what is primarily discussed as the medium of steganography. How JPEG format works in a nutshell is that it used discrete cosine transformations (DCT) to transforming 8x8 pixel blocks into 64 separate DCT coefficients. The modification of these coefficients is how steganography works. The common method for hiding data in JPEG image format data is using the least significant bits as redundant bits to hide the message in. All three of the steganography programs use this basic principle to operate, the only real difference between them is how they choose the bits as well as the order of choosing them.

Now that we have determined how steganography works, how does one detect it, and beyond that, actually decipher the message? One method that the authors remark on is using statistical analysis to examine the color frequencies of the images. Because of how steganography is implemented, the color frequency statistical properties get modified, the more data is attempted to be hidden, the more the frequency is modified in general. The authors decided to to use an extension of the statistical analysis mentioned earlier to examine images for hidden content. Because of how each of the three programs decides which bits should get used for the hiding of data, the statistical graphs gain distinct features, depending on the algorithm used.

JSteg's algorithm for choosing bits is the easiest to detect because it basically marches through, from the first one until it is done hiding the message. This means that the graph of the statistical difference will have a high probability in the beginning until it flattens out when the algorithm finishes with DCT bits. JPHide doesn't continuously select bits from the beginning so it is tougher to find. The algorithm that JPHide uses to select bits effectively ends up choosing bits that are numerically high. What this ends up doing is actually giving a very similar statistical difference graph to Jsteg, but with a rather different order of bits getting selected. The final program is OutGuess and its major difference from the previous two algorithms is how it chooses the bits to modify. Basically it uses a pseudo-random number generator to find the bits to modify; this tends to make its detection much harder. In

fact, the detection algorithm that the authors used for this article was unable to detect if a image was modified by the latest version of OutGuess.

So what happens when our intrepid authors create a web-crawler to download over 2 million images and run their detection program on them? Well, unfortunately not much; in fact 0 verifiable steganographic hidden messages were found. This is not to say that they dont exist on the internet though, just either not in the places that our authors looked (ebay and usenet) or that they were too well hidden. One must remember that even if steganography is detected on an image the password for cracking the encryption must be gained. For this our authors used a basic dictionary attack which might have been insufficient for any of the passwords they ran across. Overall this was an extremely interesting read considering that I have never heard of this techniques of encrypting messages in images.