Worms of the future

Trying to exorcise the worst

<stampf.wf@free.fr>
with inputs from
Philippe 'BooK' BRUHAT
& François MAUCHAMP

Version October 2, 2003

This is a research paper on the security (or lack of) within computer systems and ways of improvement with respect to mobile and hostile code such as worms.

In no way does the author of this paper endorse worm writers nor their activities.

This paper should be regarded as just that: a source for further research on the topic and, in the end, potential security improvement. It can be viewed as an encouragement for CIOs (Chief Information Officers) to reinforce their security through review of their security policy and quicker patching of their systems.

Table of Contents

1. Introduction

According to [Wikipedia], a worm could be defined as: *a self-replicating computer program that does not need to be part of another program to propagate itself.*

This document is an attempt at predicting the worst possible future of worms, given the current computer science possibilities.

Up to now, we've seen many different kind of worms, each new generation improving on the precedent. The fact is that all such threats, for now, have suffered from a few vulnerabilities that prevented them (much to our relief) from functioning to their full potential. Some have achieved their result to a greater extent than others, but none of them seem to have realised the greatest fear: wreaking havoc on the Internet and on Informations Systems on a global scale (although some have come close).

This document tries to look at these present vulnerabilities from a security point of view (that is, by considering the Confidentiality, Integrity and Availability of worms) and in the next chapter, how to maintain these security requirements throughout the life-span of the worm, that is to say, as long as possible. Following this, the document then attempts to provide hints on solutions that could be used in defense against new threats.

As it has been pointed out to me, other similar papers exist, one of them being [Warhol]. Surely a nice complementary reading to this paper.

2. Present worm vulnerabilities

This chapter briefly analyzes the present “vulnerabilities” in worms that prevent them from functioning to their full potential. These vulnerabilities won't be detailed too much as the path to stronger design is often rather obvious. Further improvements will be better described in the next chapter.

2.1 Approach

This document approaches the security aspect of worms through classical paths. A worm is mainly considered during Processing, Storage and Transport during which Confidentiality, Integrity and Availability must be preserved for the worm to continue to act properly.

Given that, a worm has mainly three activities:

- look for machines to infect: a worm can only counts on itself to propagate. As such, one of its tasks is to look for other machines to infect, using communication channels (mainly networks, but other potentials are InfraRed, Bluetooth, Wifi, serial, etc.)
- propagate to these machines: when a suitable destination has been found, the worm must transfer its code to it. For this purpose, it must rely on the presence of one or more vulnerabilities, either because of a bad design, or of badly configured software running on the destination computer.
- execute the payload: recent worms just replicate, but some of them (more by the past) try to cause damage to the computer on which they reside, usually at a specific date (all worms triggering at the same time).

All of these worm tasks will be analyzed in the rest of this paper.

2.2 Processing

Worms tend to protect their existence. For this purpose, they usually apply different tactics, some of them having their own intrinsic vulnerabilities:

- Intercept interrupts and check whether the worm's code in memory is being edited or not. This indeed has been a form of protection which was adopted early on by some old viruses. Quite efficient when you're not aware of it, but can easily be circumvented by debugging with care.
- Check the code's signature for modifications: this prevents in-memory modification of the code to bypass protections. But if the code can be patched in memory, so can the verification procedure.

– Encrypt the worm's code. Again, since the code needs to be decrypted before being executed, one can let the worm decrypt itself and analyze the code later by a careful use of breakpoints.

Some of these protection measures have been used by software companies to protect their code against unauthorized copies for quite a long time. The pirating industry made clear long ago that these solutions are not 100% efficient. In fact, no 100% software protection solution can be fail proof [BS1] simply because software codes have to be exposed on an owner's machine in order to be executed, over which they have complete control.

In the end, the biggest problem a worm has to face is the reverse engineering of it's code. It must attempt to avoid this as much as possible to prevent early analysis and a cure for the infection.

Another huge problem with worms processing is the relatively poor quality of their code: lots of bugs often prevent worms from functioning properly, thus resulting in poor infection and replication rates hence a short life time. We've seen lots of worms depending too much on the version of the operating system, for instance, to function properly (such as language regionalization, for instance).

2.3 Storage

Storage of code renders it vulnerable to easier analysis. Storage has often been seen as the solution to lengthen a worm's life by enabling relaunch after reboot. However, storing a worm's code on disk (or non-volatile memory) has some drawbacks:

– Storage of the worm's code on disk and configure it for automatic reload at boot time. The main vulnerability with this approach is to lay the worm's code open to possible analysis.

– Storage of the worm's code into on-disk data of other software packages by modifying it to relaunch the worm when the software is executed (this is a virus-like behavior). This is a better solution as the worm is hidden and must first be found. This is security through obscurity, however, and doesn't usually resist strong research.

– Not store the worm's code at all except in RAM. Though the least efficient solution as Availability is concerned, this probably is the better long term solution: should the computer be rebooted, the worm would die. But chances are that the computer will quickly be re-infected by its neighbors, so this is not, all in all, a definitive problem. This was the solution of choice for worms at the time of writing this paper.

As a conclusion, it can be seen that storing a worm's code has some problems as well as advantages, and both should be balanced with regard to the worm's action.

Note that we're not even talking about encryption on disk, as that would imply a decrypter with the key visibly available somewhere, thus rendering the whole thing useless. The decryption key could, however, be stored somewhere else and not be available at the moment that someone tries to reverse engineer the code.

2.4 Transport

Another activity of a worm, besides executing its payload on a computer, is to transport itself from one place to another, infecting the maximum possible number of computers. This activity, a fundamental part of a worm's life, is also at the root of its main problems:

– When a worm has started to infect computers, it also starts to replicate and transport itself on networks. Consequentially, traffic exponentially increases with each new generation of worms, causing network congestion and raising IDS alarms. This results in the worm being eventually detected.

– As a worm has to travel across networks it will, sooner or later, trigger an IDS that will start recording traffic and result in the worm's code being saved on disk. Once this is done, an anti-virus signature can then be crafted for that worm which could then be detected and blocked (without talking about analysis). The current solution to this problem is to use a polymorphic code decrypter when the main code is encrypted.

2.5 Author's protection

One fundamental problem for virus writers these days is the law. Indeed, quite a number of authors of the recent most effective worms have been chased and caught: everybody knows that your activity can very often be traced back up to your Internet connection. This is simply because you have to connect from somewhere, hence have a login, a password and a phone number to access the network (or cable connection, DSL line, etc.)

Of course, virus writers can use relays, anonymizers and all manner of gadgets to masquerade their true identity and origin, but there are still other means (at least statistically through studying the propagation of the infection and by studying the virus's code and the hackers community to find the author). This most often also implies a form of cooperation between network administrators and international law enforcement agencies.

3. Possible worms evolutions

This chapter tries to analyze some possible worm evolutions in terms of technology in order to improve their security, thus rendering their detection and/or eradication more difficult. The following chapter will then try to investigate possible protection measures that could be crafted against such improvements.

3.1 Processing

Regarding processing, we've highlighted the two main problems with worms' code:

- a need to protect itself from reverse engineering;
- a lack of quality.

In addition, we present further improvements to worm code, not related to actual vulnerabilities.

3.1.1 Protection from reverse engineering

There are most likely many possible solutions to this problem. We're trying to highlight some of them below.

3.1.1.1 Prevent debuggers from viewing the code

As there are kernel modules (dynamic kernel patches or dynamic libraries) that can render processes, files or network connections invisible to other processes, one can imagine a solution where some piece of memory would not be easily viewed (either by forbidding it from being read without the right privilege or by placing it outside the program address space, for instance). One needs to investigate the possibilities of a Pentium's (and other processors) capabilities within this domain.

3.1.1.2 Detect debuggers in real time

The most evident solution is to try to detect a running debugger in real time and block the computer if this is the case.

This solution is already implemented in some Copy Protection schemes to avoid reverse-engineering and removal of the protection.

The reader is advised to check the Internet for solutions on that topic, but we can already give a few hints:

- intercept debugging traps and check registers: if one of them points to the worm's code at the time of interruption, block the computer;
- intercept frequently raised interruptions: clock, mouse, disk I/O, etc.: each time, check registers as above.

3.1.1.3 Prevent debuggers from debugging

Preventing debuggers from functioning properly is mainly a matter of preventing the user from easily tracing the worm's code. We can imagine the following scenarios:

- encrypt the code. For better efficiency, the code may be chunked and every piece of it encrypted with a different key or algorithm.
- Use system parameters as decryption keys (or even better, a hash of different parameters): if system configuration changes (debugger trapping, interrupt for

instance), the code can't be decrypted (provided the interrupt handler is included in the hash).

– Use concurrent threads to handle pre-decryption and post-re-encryption or deletion of the code. If the computer gets slowed down, desynchronisation occurs and the code fails (or the computer may crash if the worm's code runs with the maximum of privileges such as Intel x86 ring 0 for instance).

– Have hashes being computed by threads concurrently and used asynchronously. With asynchronous threads handling decryption and encryption ahead and after the Instruction Pointer, one could imagine an ever changing code. Changing the speed of the code execution through debugging, will stop the worm from functioning, remaining encrypted and crashing the computer.

– Time protect the encryption by including the time in the decryption key: if it's not the right time, the code can't be decrypted.

– Store decryption keys somewhere else, possibly on an Internet connected machine under control of the worm's author. Or use chatting networks to get the key (real-time chatting networks [such as IRC, ICQ, etc.] are quite dangerous for the worm author's own privacy, whereas non-connected chatting networks [such as usenet news for instance] are quite convenient for this purpose).

– Have multiple processes running on multiple machines (not necessarily the same architecture) cooperating to build the code or decrypt one another (by transferring decryption keys across processes). Parallelism might be a new way for worm code to work.

Of course, these protection measures are not 100% secure. They can't be. But they can be a real nuisance for an anti-virus company that would need to debug a worm's code and implement appropriate patches to provide protection against it. More time for the worm would mean more time for infection.

3.1.2 Improving code maturity

This can only be achieved by developers through maturity and by taking time to test and develop their code.

Through maturity disappears frenzy, and thus better code can be achieved.

Chances are that the more divergence between governments and hackers/crackers there is, the deeper the hacker's involvement in gray to black hat activities will get.

3.1.3 Keep a low profile

Keeping a low profile for a worm means reducing immediate and obvious actions right after infection to avoid early detection. The longer a worm manages to rest discreet, the higher the number of hosts it will be able to infect prior to detection.

Later on, all worms may wake up on a single signal, through a worm-net for instance.

Alternatively, each worm may wake up at a random time, way later.

One could imagine a discreet payload or one that comes along with the signal to wake the worm (indeed, this solution has been already implemented by the latest big worm that we've seen: [SoBig.F], though not with exactly the same scheme).

Finally, one could imagine worms in a defined environment (company) that all wake up at the same time when the system administrator tries to deactivate one of them.

3.1.4 Adaptation to environment

To be successful and live long, a worm has to adapt to its environment. This means updating to new vulnerabilities and even new architectures.

Possible solutions are:

– connect to Internet places to download plug-ins. These must be verified for a valid signature in order to avoid installing a killer plug-in ... Also, the worm will need to take care while using the standard Internet connection method of the computer on which it resides, to avoid being spotted (use the host's proxy

configuration or replay authenticated packets through the proxy, for instance)

- use various sources for updating: there are usually more than one communication channel with Internet. One can expect any of the following:
  - web access (with or without a proxy, with or without authentication);
  - SMTP (or other protocol) mail access;
  - news (NNTP) access;
  - the path the worm itself took to come from (i.e. download other code from the remote computer it came from, in a kind of worm-net);
  - other covert channels above IP: sequence numbers, flags, timings... (although not all of them might be available through a firewall);
  - direct access, which opens a wide range of possibilities among which: IRC, P2P networks, possibly with anonymity (see [6/4] for instance or [FreeNet]); such communication channels should mainly focus on avoiding a single point of command (to kill the worm or to find all of them). One could also consider it as a means of gathering statistics or passing messages from the author to the worm population.
  - IPSec (on top of IPv4 or embedded on Ipv6);
  - (SSL) tunnels through HTTP/S proxies which allows for any kind of embedded protocol inside the tunnel;
- use multi-architectural code: for one processor (i686 for example) the initial code acts as a 'jump' command directly to the worm's dedicated code whereas on another processor (say, SPARC) it simply executed an instruction without side-effect. The following part of the code being the significant part related to this processor. This has already been used for cross-platform buffer overflows.

3.1.5 Payload

The very first viruses and worms in existence often used to carry a destructive payload. That's rarely the case today, but this may change.

We could imagine the following examples of payloads, however it might be noted that the options for creativity in this domain are more vast than in any of the other chapters of this paper.:

- destructive payload: wipes hardrive, re-flash BIOS with zeros;
- re-flash BIOS to incorporate the worm's code;
- edit documents in subtle ways: lightly edit dates, change numbers (significantly less obvious to the naked eye): imagine the consequences for contracts, medical registers, banking applications, etc. But one might also wonder if an attacker would take the risk of inadvertently changing his own medical records or, more significantly, directly kill people because of his worm's effects (in case of hospitals getting attacked by the worm). On the other hand, would terrorists care about that? Wouldn't they be on the look-out for just such a destructive payload?

3.1.6 Protecting against protections

For all of the solutions presented above, it can be useful for the worm to regularly check whether its measures of self protection are still present and working. Cross checking is even better: this can be done through cryptographic signatures for instance. Simple CRCs are not enough as they can be recomputed by the reverse engineer.

Again, not all protection measures are 100% secure, but they can be hard enough to circumvent to let the worm “sufficiently” propagate.

Some paths that may be interesting to follow on this subject include:

- Zero Knowledge Proof [] to check that the code is correct (proof without revealing the code);
- Identity-Based Encryption [IBE] might be worth a look by using the code as the public key and using an external service (not under control of the infected

machine) to provide the decryption key;

– Elliptic Curve Cryptography [ECC] is also interesting for its rapidity.

– Exploit multiprocessor computers to have multiple part of the worm running concurrently and checking the status of each other at random.

3.2 Storage

As we've said before, storing its code somewhere would probably be the worst thing that a worm could do today. It's an invitation to reverse-engineering. It's like an autopsy: you can see whatever you want and take your time doing so: the body is dead and won't stand in your way.

A few notes on EPROM and programmable BIOS: if the worm infects the BIOS by means of re-flashing, it can gain more life than expected, or at least until anti-virus vendors figure this out and reverse-engineer the PROM. It is also, however, a form of storage as well with the corresponding vulnerabilities (see 2.3 - Storage).

One solution could be to hide the worm's code on another platform: use Unix machines to infect Windows, and Windows to infect Macintosh (for instance). Once a platform has been cured, chances are that the system administrators will look for the worm on other similar platforms, when in fact it's hidden somewhere else. See point “3.1.4 - Adaptation to environment”, about multi-architectural code. Similar solutions involve LAN attached hard disks, open web sites, FTP sites, open network shares, etc. One can also imagine the use of a PDA to host the code intended to infect desktop computers during synchronization.

If the worm's code is stored on disk, it could be encrypted and the decryption key could be stored/available somewhere else, possibly as a temporary measure. One could imagine a place that renders the key available only at predefined hours, with worms regularly polling the web sites. Or another process, running on another computer (possibly on another architecture) that sends the key over the network at predefined intervals: the worm would have to sniff the network to get it. The code sending the key would conceal its origin by trying to masquerade as another host, spoof source addresses for example (MAC spoofing is especially useful to avoid easy detection). It could take hours or even days for the key to be sent.

3.3 Transport

One of the main activities of a worm, hence it is also a place for considerable improvement.

Network flows can often be easily spotted either because there are too many of them, they are too big or use standard protocols (UDP, TCP/IP) with non standard parameters (such as non standard port, easily blocked by firewalls, for instance).

The following possible improvements to worms quickly come to mind.

3.3.1 Discretion

– Avoid too many connections: it's probably better to send one big packet than many small ones. This, however, needs to be checked against statistics on a network full of worms uploading themselves across the globe. The idea here is that IDSes often neglect packets when they are only few in numbers and only raise alarms on crowd detection, to avoid too many false positives.

– Try to blend into legitimate traffic: if using HTTP as a transport protocol, then connect through the proxy as a normal client would do.

– Spoof source addresses whenever possible and sniff for responses.

– A tool (or badly coded piece of software) had been causing some trouble on Internet during the summer of 2003, sending source spoofed packets that were TCP SYN with a window size of 55808. Supposedly a proof of concept (although defective) mapping software package, it could also have been some form of command for a discreet worm, with data being encrypted in the packet's payload. This is an interesting approach to sending packets to an installed worm base without directing them to the worm (hence revealing it). The worm has to sniff the network to get its commands.

3.3.2 Signature

The main problem of a worm in transit is its network signature in IDSes. The signature can be made of:

– a stream of bytes

– a behavior

Trying to avoid both is the worm's best move.

Possible solutions for avoiding these problems is to:

– use lots of different transport solutions, like different protocols, encrypt traffic, use dynamic ports, etc.

– Another would be to use polymorphism.

Another approach is to use IDS signatures for other signs of attack, to flood IDSes. By sending lots of signatures, IDSes will start to work overtime. In a high traffic environment, this can result in either:

– IDS missing some packets;

– the security administrator in charge of the IDS being unable to monitor all the alarms at the same time (and not knowing which ones are for real).

3.3.3 Polymorphism

Polymorphism is a protection by which a code changes itself to avoid recognition and signature making. We can envisage different kinds of polymorphisms:

– using encryption: the code is encrypted with a different key each time it gets transferred on the network. The main obstacle is that the decryption code must be made readable and from this, a signature could be made.

– Using code mutation to do the same with different assembly operations. For instance, to put 5 in a register could be as simple as that, or putting 10 and subtracting 5, or putting 4 in a place, reading that place in the register then adding 1, etc. That could be used for decoding a portion of the code mentioned above. See [ADM] for an example of NOP mutations (this would need to be adapted for other instructions, possibly respecting the “no zero byte” constraint).

– In case a smart debugger is created that analyzes code behavior to bypass the polymorphism described above, one could imagine the separation of the code into blocks of independent actions and have them executed in as many different orders as possible. Checkpoints would be needed at some time, but that could change the order in which the code executes, without changing the overall result of operations. See Illustration 3.1 - Polymorphism.

[Phrack#61] implements an improved polymorphism in order to avoid the spectrum analysis of traffic. This has yet to be included in a worm's code, but given that the tool is available, it could be just a matter of weeks until it's put into practice.

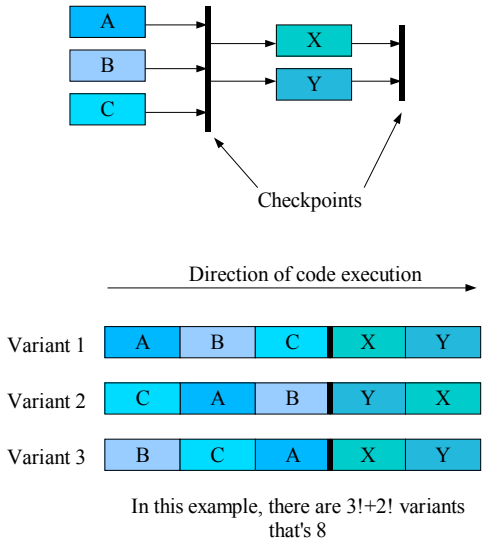

Illustration 3.1 - Polymorphism

3.3.4 Avoid network congestion

Another problem that worms face is often their huge impact on network bandwidth. A high rate of infection means a high number of worms roaming on networks and thus negatively affecting bandwidth.

This is sometimes the sign of something bad going on.

A worm could avoid this problem by decreasing its expectations in terms of infection rate, in favor of a prolonged life-time. This would imply that each generation of the worm is slower to infect further computers.

The rate of infection would be calculated on the percentage of hosts vulnerable to infection and the number of other hosts each infected computer is willing to infect. Indeed, this has already been researched in [AAWP].

3.3.5 New moving paths

Given the latest technology available, new paths of infection arise:

- WiFi communication channels: while it used to be safe enough to avoid connecting to unknown networks, WiFi technologies have a tendency to make the machine onto which they are installed adopt a state of permanent connection (or connect to whatever network is available in the vicinity of the computer). Given the relative poor workstation security, this provides a great opportunity for worms to infect neighboring computers.
- Infrared beams: usually only used for serial communication, [IrDA] can allow the direct transfer of objects to the remote peer (try placing a PalmOS device in front of a Windows laptop: they both automatically connect together).
- Bluetooth: super small radio connections with built in magic that allows the transfer of information between the communicating devices. As security is often relatively complicated to configure (when configuration is even available), this also provides a potentially easy way of infecting devices.

Finally, imagine a worm that uses other platforms to host its code and new moving paths to infect computers. Such as a worm moving from PDA to PDA (or phone) through IrDA or Bluetooth and infecting the computers that it connects to, then again moving from computer to computer and infecting other PDAs. The more infecting vectors there are, the more difficult it will be and longer it will take to get rid of the worm. Did I mention the permanently connected phones and PDAs that use [I-Mode]?

3.4 Author's protection

Now, the last point left is the protection of the worm's author. If you had written a worm implementing the preceding functionalities, would you really want police forces from all over the globe after you? I'm sure you'd prefer to remain as anonymous as possible.

Fortunately for you (unfortunately for the police), one of the latest breakthroughs in technology can help you: WiFi. More precisely, free, anonymous WiFi.

Nowadays, just by roaming around any big town, one can find dozens of (if not more) freely accessible connection points, either because of mis-configured equipment or because the place is truly letting everyone use it. It's just a matter of using the facilities of a cybercafé without entering the premises. Nothing to pay, no risk of being video-recorded. Nothing. Consider WiFi communities for example, restaurants and cafés offering free WiFi access (McDonald's for instance)...

Coupled with a small worldwide community of pirates that launch infections throughout the world, this concept starts to sound quite frightening.

Furthermore, it can often prove difficult to find the first computer(s) to have been infected. By using poorly configured WiFi Access Points, an attacker (or community of) can infect different companies across the planet and consequentially greatly increase the initial rate of infection. Having multiple, simultaneous sources of infection also greatly increases the difficulties faced when attempting to trace the infection back to it's origin(s).

3.5 The worst possible picture

Well, what could be the worst possible case when we try to assemble all of the preceding points? Here is a glance, your imagination may vary:

There could be a sort of generic worm engine, providing meta-functionalities such as polymorphism, exploitation of vulnerabilities, transportation and payload execution, plus an additional update mechanism. All of them, including the update mechanism, would be implemented as replaceable and sequence-able plug-ins. That means, you could replace the meta-functionality, add or remove plug-ins, possibly having more than one available at a time. Which means the worm could update itself to reflect the latest security vulnerabilities available. While the first version would transport itself using, for instance, a web vulnerability, the next generation could use the same vulnerability along with a new one (RPC for instance) plus a communication channel across Kazaa (which would then be modified for eMule), then gets its polymorphism engine changed to increase its entropy, etc. The update mechanism would naturally be capable of knowing when an exploit is outdated because it is not efficient enough in attacking computers with regards to other more recent ones: this would allow the worm to control the size of its code.

Could it be that with a sufficiently quick code writer base, the worm would get updated so rapidly and often that it would defeat any possible destruction by anti-virus companies, possibly running indefinitely (or at least as long as worm code writers keep updating it and don't get caught), always exploiting the latest holes?

4. Protection from doom

After exposing the possible worm improvements above, this chapter attempts to identify current ways of protecting against such threats, as well as give hints on future research that may need to be conducted to protect from what could be described as "intelligent" worms.

4.1 Keep security up to date

That probably seems an obvious suggestion, but some people still just forget about it.

Unfortunately, lots of system administrators simply don't have the time to keep up with the high number of patches and new versions being released every week or month. Obviously, some fundamental efforts should be made in this area before even considering moving on to some of the more specific solutions as suggested.

As patch application is often difficult because they need to be validated before going into production, one could imagine a rollback possibility so that in case of a patch failure, the system could be quickly restored to its previous state in order to reduce the down-time of a machine. Windows XP [WXP] implements such a concept of rollback.

Another useful initiative to be used in conjunction with patches is the Application Vulnerability Description Language initiative [AVDL]. Carefully used with IDSes, firewalls and patch management, it could be an incredible tool to:

1 – check for vulnerabilities,

2 – patch vulnerable systems

3 – protect unpatched systems from attacks

4.2 Processing

4.2.1 Protect from infection

This is the main action that should be carried out: protecting from infection.

As far as stages in security functionalities are concerned, this is the first to consider: Prevention. Detection comes later. As for Recovery, there's probably not much that can be done other than the current implemented solutions: backups and continuity plans.

4.2.1.1 Change OS kernels

One can consider modifications in Operating System kernels. A few come to mind when dealing with processes.

Non executable Stack

Just when data structures for a new process are being set up, it can prove useful to mark the stack space for this process as being non-executable. This solution is quite effective in rendering most buffer overflows ineffective. Indeed, this has already been successfully implemented in Linux (in OpenWall, see [LNESP]) and Solaris (option `noexec_user_stack` in `/etc/system`). Few software packages should be impacted by such a measure.

Non executable data memory

By transforming the data portion of software code as being non executable, one can further increase the protection of non executable stack (see above).

Both solutions involve setting a flag during memory allocation to prevent execution by the processor. Not all processors may allow such a solution, however. There have been some messages exchanged on this subject in the ia64 Linux mailing-list [NEDM].

Read only code

This solution is to mark the code in memory as being non writable. This is just in case the software is tricked into loading, or easily changing a new portion of code to change its behavior. This has already been implemented in lots of operating system.

Signed software code

Another solution could be to digitally sign all software code (whether in extensions such as DLLs, shared libraries, kernel modules or whole executables) with a private key. Non-signed code is denied access and/or execution or constrained to sandboxes.

This has already been partly implemented by Microsoft with Authenticode for ActiveX. (See [Authenticode]). This solution could be improved by requiring a Trusted Path from the kernel to the user before accepting a new signing key.

4.2.1.2 Change compilers

Different ways exist to improve security at compilation time. However, this would require a full recompilation of all software running on a system, and that all software installed later be compiled with the same measure of protection as well. Gentoo Linux is one such distribution ([Gentoo]). Once the compiler is patched, all can be recompiled.

Bound check variables at compilation time

The first solution is to check variable boundaries during code compilation and refuse overflows. This might need a change in language definitions (C family for instance) to enforce this and a complete redesign of buffer handling (auto grow, fixed chunks...) ADA as a language does this kind of checking and could be a source of inspiration [Ada].

Remove dangerous functions from libraries

A few programming functions (most often based on the C language) are vulnerable to buffer overflow. While this is not the only source of worm infections, removing these functions or replacing them with other ones could suppress a whole series of infections.

This would have a considerable impact on software development however, and is not very practical. But when considering Open Source work such as that which is used in [OpenBSD], this is surely one way forward ("*Only one remote hole in the default install, in more than 7 years!*" as of 2003).

Another solution could be to carry out some special checking when one of these functions is used. See [LibSAFE] for instance.

Don't allocate variables on stack

Another radical solution would be not to allocate subroutine variables on the stack. Keep the stack for that which it was created, that is: storing Instruction Pointer's return addresses. Allocate variables somewhere else in the processes memory space.

User stack integrity checkers

Also called *canaries*, in reference to the birds used in mines to detect firedamp and avoid consecutive explosion: should the canary die, it would signal the presence of firedamp in the air and all miners would stop working and evacuate.

The same principle applies to computer science: the stack is marked with canaries each time a subroutine is called for. The end of all subroutines is specifically crafted (at compilation time) to check canaries on the stack before calling the return address. If a stack-allocated buffer became overflowed and destroyed or altered the canary, the program gets signaled. [Immunix] implements this solution through StackGuard.

4.2.1.3 Change computers

By designing a secure computer right from the boot sequence, one could imagine more robust security solutions. This is just what ~~Palladium~~ Next Generation Secure Computing Base ([NGSCB]) is about. A specially crafted hardware component, implementing cryptographic functions that check codes and configurations. Should one of those not be compliant, actions could be taken to prevent them from misbehaving.

One can also imagine new ways of "understanding" code through heuristics and detect that it is self modifying: a request for special authorization from the user through a trusted path would be needed to accept such behaviors.

4.2.2 Prevent software from acting when infected

By using operating systems that implement privileges, it is possible to control what can be done by a piece of software. Without the specific right, the software cannot do much harm.

If one imagines a way for an application to request a right to carry out an action, a sort of trusted path could be put in place for the user to do so, after verification of the application's integrity. See the section on NGSCB above. Such solutions already exist: [TrustedSolaris], [seLinux] and [RSBAC].

4.3 Storage

There are very few improvements that could be made on this topic, but we can give the following hints:

– software executables should never be writable for the user. Although already the case on Unix computers, Windows has still some improvement to make in this domain (although the latest versions have almost totally suppressed the concept of "home user can do anything").

– Sign software code on disk, configuration files, etc. Require the user to enter a password to unlock the signing key (again, through a trusted path).

4.4 Transport

4.4.1 Understand code semantics at firewall level

As a result of strong polymorphism and encryption, signature based detection is hard, bordering impossible. Introduce new ways of "understanding" code through heuristics and detect whether it is self-modifying: a special request for authorization from the user through a trusted path would be required for such situations. One way to go would be to reconstruct the actions carried out and not the way that they are achieved (not the instructions). This would probably involve detecting pseudo NOPs with respect to registers and actions of the code being analyzed.

4.4.2 Detect uncommon behavior

IDSes should detect unusual packets that differ from traditional ones: make signatures based on traffic shapes rather than content. This is called spectrum analysis and has just been described in [Phrack#61] with a new polymorphic code engine to counter it.

IDSes could also communicate with one another to increase the sensitivity of detection: one packet may not trigger an alarm (to avoid too many false positives), but one packet received by many IDSes may be a sign of wide attack (either attacker or worm), coordinated

or otherwise.

For small companies with only one Internet access point, it may make sense to offer traffic shaping comparisons between different companies. Other concerns then arise: how to preserve confidentiality and anonymity?

4.5 Author's protection

Well, it's hard to act on this subject, but one of the (incomplete) solutions might be to force all Internet entry points to be authenticated. Non-authenticated entry points might, one day, be considered as SMTP open relays are today, with dedicated blacklists. That won't prevent spoofing though. Some sort of mandatory source-routing might be a solution to trace back the packet to its entry point in real time.

Unfortunately, this is barely enforceable today. China has tried to control their citizens going on Internet, but it's known not to be very efficient.

5. Conclusion

Well, that's all for the nightmare. In the end, what is left and what could be done as an emergency measure?

I'm afraid there's not much that can be done to approach 100% efficiency. Not now. Not even close to such a figure.

Probably one of the most promising security measures would be traffic shaping IDSes and communication between different ones. They still need a lot of improvement to prevent false positives, because false positives tend to bore administrators who then end up not listening to alerts anymore.

Artificial Intelligence might be the way to go, along with cooperation between IDSes (especially inter-company co-operation).

In this respect, I look further to the work on IDSes and Honey nets [HNP] as means of distributed detection.

Despite all the polemics around the Microsoft initiative (NGSCB), it might be the most promising solution. Too bad nothing will be available before the next generation of worms: it's easier to act bad than good. Maybe the Open Source community could start a project to counter Microsoft's initiative and the evil they see in it (be it real or not) to offer an alternative to Microsoft's implementation as soon as possible. A good starting block could be the code donated by IBM to access the TCPA chip [TCPA].

A word on vulnerability full disclosure

Full disclosure of vulnerabilities is surely a way of facilitating the task of exploit writing. On the other hand, imposing a total blackout on vulnerability discovery and disclosing it only to the vendor (or publisher) of a piece of software is surely a way of; 1- not inciting hackers to discover holes, and 2-having some hackers keep them for their group of friends to carry out their own exploits.

As such, the author of this paper sees Responsible Vulnerability Disclosure Process [RVDP] as a good method that hackers could use to improve Internet Security. As funny as it is, they all say that their work aims at improving security, while full zero day disclosure does just the opposite. RVDP is just their chance to prove that they can mature and behave as they say they will.

Good old solution: patch, patch, patch

Last word: patching a system as soon as a solution has been found to a security vulnerability has always been the best solution to avoid security problems. More work in this domain is probably the best move people could make in order to avoid being in big trouble, until other solutions are designed. Oh, and keep your firewalls well configured as well.

And keep your users informed about worms and not opening suspicious email attachments.

6. References

- [6/4] the 6/4 protocol: a P2P protocol ensuring privacy and anonymity to its users. http://www.hacktivismo.com/
- [Ada] Ada programming language with variable bound checking. http://www.adahome.com/
- [ADM] ADMmutate, shellcode mutation engine http://www.ktwo.ca/security.html
- [Authenticode] Microsoft Authenticode technology. http://msdn.microsoft.com/workshop/security/authcode/authenticode_node_entry.asp
- [AVDL] Application Vulnerability Description Language http://www.avdl.org/
- [AAWP] Modeling the spread of active worms http://www.hackbusters.net/LaBrea/AAWP.pdf
- [BS1] Bruce Schneier's crypto-gram newsletter of May 2001: “The Futility of Digital Copy Protection” http://www.counterpane.com/crypto-gram-0105.html#3
- [ECC] Ellipctic Curve Cryptography http://www.google.com/search?q=elliptic+curve+cryptography
- [FreeNet] is another privacy and anonymity network. http://freenet.sourceforge.net/
- [Gentoo] Gentoo Linux: source only Linux distribution http://www.gentoo.org/
- [HNP] The honey net project. http://project.honeynet.org/
- [IBE] Identity-Based Encryption http://crypto.stanford.edu/ibe/
- [Immunix] Linux distribution with canaries in stacks http://www.immunix.org/
- [I-Mode] I-mode http://www.nttdocomo.co.jp/english/p_s/imode/
- [IrDA] Infrared Data Association http://www.irda.org/
- [LibSAFE] Avaya's open source library for protecting the critical elements of stacks. http://www.research.avayalabs.com/project/libsafe/
- [LNESP] Linux Non-Executable Stack Patch http://www.openwall.com/ (incorporated in the OpenWall project) and an exploit that bypasses this protection: http://www.insecure.org/sploits/non-executable.stack.problems.html
- [NEDM] Non executable data memory for Linux ia64 discussion https://lists.linuxia64.org/archives/linux-ia64/2002-January/thread.html#2726
- [NGSCB] Next Generation Secure Computing Base by Microsoft https://www.microsoft.com/resources/ngscb/default.mspx
- [OpenBSD] Free multi-platform 4.4BSD UNIX-like operating system with proactive security and integrated cryptography. http://www.openbsd.org/
- [Phrack#61] Phrack #61, Polymorphic Shellcode engine http://www.phrack.org/show.php?p=61&a=9
- [RSBAC] Rule Set Based Access Control for Linux http://www.rsbac.org/
- [RVDP] Responsible Vulnerability Disclosure Process. http://www.globecom.net/ietf/draft/draft-christey-wysopal-vuln-disclosure-00.html
- [seLinux] NSA seLinux http://www.nsa.gov/seLinux/
- [SoBig.F] Description of the SoBig.F worm http://www.wired.com/news/infostructure/0,1377,60150,00.html
- [TCPA] IBM donated linux device driver code to access the TCPA chip http://researchweb.watson.ibm.com/gsal/tcpa/
- [TrustedSolaris] Trusted Solaris environment http://wwws.sun.com/software/solaris/trustedsolaris/
- [Warhol] Warhol Worms: The Potential for Very Fast Internet Plagues http://www.cs.berkeley.edu/~nweaver/warhol.html
- [Wikipedia] Wikipedia, the free encyclopedia http://www.wikipedia.org/
- [WXP] Windows XP with rollback OS capabilities http://www.microsoft.com/windowsxp/default.asp
- [ZKP] Zero Knowledge Proof http://www.google.com/search?q=zero+knowledge+proof