Exploit Hijacking: Side-Effects of Smart Defenses

Costin Raiciu Mark Handley David S. Rosenblum
Department of Computer Science
University College London
{c.raiciu|m.handley|d.rosenblum}@cs.ucl.ac.uk

Abstract

Recent advances in the defense of networked computers use instrumented binaries to track tainted data, and can detect attempted break-ins automatically. These techniques identify how the transfer of execution to the attacker takes place, allowing the automatic generation of defenses. However, as with many technologies, these same techniques can also be used by the attackers: the information provided by detectors is accurate enough to allow an attacker to create a new worm using the same vulnerability, hijacking *the exploit. Hijacking changes the threat landscape by pushing attacks to the extremes (target selectively or create a rapidly spreading worm), and significantly increasing the requirements for automatic worm containment mechanisms. In this paper, we show that hijacking is feasible for two categories of attackers: those running detectors and those using Self Certifying Alerts, a novel mechanism proposed for end-to-end worm containment [4]. We provide a discussion of the effects of hijacking on the threat landscape and list a series of possible countermeasures.*

1 Introduction

Recent advances in the defense of networked computers use dynamic run-time instrumentation of binary executables to track tainted data, and can detect attempted break-ins automatically [14, 13, 9, 5, 11]. These techniques identify how the transfer of execution to the attacker takes place, allowing the automatic generation of defenses.

However, as with many technologies, these same techniques can also be used by the attackers. Consider an attacker who has already compromised hundreds of desktop machines, perhaps using an email virus. He can then run an instrumented server on these machines. When a new exploit for this server software is discovered, his aim is to discover the exploit early. These detectors provide complete information about the exploit. In many cases this is enough to *automatically* generate a new worm using the same vulnerability. In effect the exploit has been *hijacked*.

Exploit hijacking changes the threat landscape. An attacker with a new exploit has two choices: target very selectively so as not to risk triggering a detector, or compromise as many hosts as possible via a rapidly spreading worm—a flash worm [15]. The speed of the worm matters greatly. He knows that his competitors may in turn hijack his worm; the fastest spreading worm will win. Competitive pressure will result in only very targeted attacks, or worms that compromise the entire vulnerable population in seconds. It is likely that no attack between these extremes will survive.

One of the most promising technologies to defend systems against worms and other software exploits involves the use of Self-Certifying Alerts [4]. These are descriptions of exploits that contain enough information to allow an end-host to automatically verify whether a vulnerability really exists in its software. SCAs can be created automatically using taint-tracking detectors, and distributed to potentially vulnerable hosts. Each host can safely check the SCA locally to determine if it is vulnerable, and if so, it can automatically generate a filter to avoid being compromised. Using a peer-to-peer network, it is possible to distribute SCAs rapidly, checking them at each hop along the way to avoid propagating false alarms. The hope is that most vulnerable hosts are alerted before they can be compromised.

However, as with detectors, there is a downside. We have developed proof-of-concept code that demonstrates that many SCAs contain enough information to allow the generation of new worms using the same vulnerability.

This paper details the cat and mouse game between the automated attackers and automated defenders that now seems inevitable. We show that hijacking is feasible in Section 2. We discuss and evaluate the impact of hijacking on the threat and defense landscape in Sections 3 and 4. In Section 5, we list possible defense strategies. We summarize our arguments in Section 6.

2 Exploit Hijacking

Finding software flaws and turning them into exploits is not a trivial task, as it requires a great deal of knowledge and creativity. In contrast, manually crafting a new exploit from an existing one is significantly easier and has even been used by the creators of infamous Internet worms such as Blaster and Slammer. Exploit code was publicly available in both cases weeks before the worm outbreaks; creating the worms was only a matter of somebody modifying the exploit code. Therefore, given an existing worm (i.e. its attack messages), it is usually easy to manually craft a new worm—to hijack it. However, manually hijacking a worm will usually bring little benefit: by the time the hijacked worm is available, the initial worm has already infected most of the susceptible population. As many worms patch the vulnerability they use, the manually modified worm will

Service: Microsoft SQL Server 8.00.194
Alert type: Arbitrary Execution Control
Verification Information: Address offset 97 of message 0
Number messages: 1
Message: 0 to endpoint UDP:1434
Message data: 04, 41, 41, 41, 41, 42, 42, 42, 42, 43, 43, 43, 43, 44, 44, 44, 44, 45, 45, 45, 45, 46, 46, 46, 46, 47, 47, 47, 47, 48, 48, 48, 48, 49,49,49, 49, 4A, 4A, 4A, 4A, 4B, 4B, 4B, 4B, 4C, 4C, 4C, 4C, 4D, 4D, 4D, 4D, ...

Figure 1: Arbitrary Execution Control SCA for Slammer

have little impact. Interestingly, the same problem exists in defenses against worms: if signatures are manually generated, they come too late to stop the infection of the spreading worm. To have much impact, both hijacking and defenses must be automated.

2.1 Hijacking Using Host-Based Detectors

Instrumented software designed to detect attempted break-ins works by keeping track of tainted data (data derived from messages received from the network) as it is used by the program. If this tainted data is executed or used as a jump address, then an exploit has been detected [14, 13, 5, 11]. By tracing back the tainted data to its origin in the message logs, the detector finds the message that contained the exploit. Normally this would be used to generate an alert or patch, but an attacker can use it instead to generate new malicious code that uses the same vulnerability.

All the attacker has to do is to paste his worm code over the original payload, as determined by the detector. A version of hijacking for the good (to create automated anti-worms) was proposed by Castaneda et al. [3]. The techniques provided there are further demonstration that hijacking using detectors is feasible.

2.2 Hijacking Using SCAs

A Self-Certifying Alert is a message that describes a specific exploit of a vulnerability in enough detail that the existence of the vulnerability can be automatically verified. Three types of SCA are detailed in [4]:

- *Arbitrary Code Execution SCAs* describe how to inject and execute code in the vulnerable program.
- *Arbitrary Execution Control SCAs* show how to divert a program's execution flow to a particular memory location.
- *Arbitrary Function Argument SCAs* show how to supply parameters to arbitrary function calls.

An example of an Arbitrary Execution Control SCA from [4], is provided in Figure 1. The SCA tells the host that placing an arbitrary address at offset 97 in the supplied message and sending it to an instance of SQL Server version 8.00.194, will cause the program to jump to that address. This information is used by a verifier to check the existence of the vulnerability.

As techniques to exploit the various types of SCAs are different, we separate the discussion for each type of alert.

2.2.1 Arbitrary Code Execution SCAs

Arbitrary Code Execution SCAs are easiest to use in automatic exploit generation with high likelihood of succeess in the wild. When an SCA arrives that describes a arbitrary code execution vulnerability, the hijacker merely writes the exploit code at the offset specified in the SCA. Assuming that the exploit code is small enough and general enough to work on multiple platforms, the hijacker can now launch the worm in the wild.

We tested this technique for two existing worms, Blaster and Slammer. Rather than generate a new worm, we used existing exploit code that gives the attacker a remote command shell [2]. The code is independent of the Windows variant, is reasonably small (332 bytes), and contains no null characters so is usable in strcpy-like overflows. To simulate SCAs, we used the publicly available code for Slammer and Blaster and identified the address of the shell code inside the attack messages. The process of hijacking the worm was reduced to overwriting the original exploit code with our shell code. Hijacking worked on first attempt, without any debugging, for both Blaster and Slammer.

2.2.2 Arbitrary Execution Control SCAs

Leveraging Arbitrary Execution Control SCAs is a bit more complicated than code execution SCAs. The SCA tells the hijacker how to direct the vulnerable software to jump to any specified address. However, the hijacker is not told how to place exploit code at a known address inside the process's address space.

Automatically mapping the exploit code at a known address does not appear to be easy, but there are at least two basic ways to do this. In both cases, our approach is to build offline a database that describes how to map data at specific known locations.

The first approach assumes the arbitrary execution control is due to a stack-based buffer overflow. The attacker places the exploit code immediately after the (overwritten) return address in the attack message. To jump to the exploit code, he needs to find some code in the vulnerable program that executes a "jmp esp" instruction (or an equivalent). Using a tool called findjmp [1], we find such an instruction in kernel32.dll at offset 0x7C82385D, for Windows XP SP 2. As kernel32.dll gets loaded with every Windows executable, the hijacker can use a debugger to find the base address of kernel32.dll in the vulnerable software's memory space. Slammer and Blaster, using stack-based buffer overflows, can both be hijacked in this way. We note that this offset is OS dependent and therefore multiple database entries must be maintained per software product, corresponding to different OS versions.

Our second approach is to use the services provided by the vulnerable process to map code at predictable locations in memory. Creating a database of memory invariants in the target software is not an easy task, but it is often feasible.

This approach has more coverage, being applicable to the majority of arbitrary execution control SCAs.

For concreteness, let us consider Microsoft's IIS 5.1 Web Server. The server is multi-threaded, so data mapped into one thread is visible to the other threads. Using HTTP, we can place arbitrary code into memory by encoding it into the resource name, as multipart or form data, or as HTTP headers. For IIS, we found the following invariants:

Heap Addresses. For idle servers, we can use predictable heap addresses to map data in memory (examples include 0x71cb6f, 0x11ce8f, etc.). Reliability can be increased by sending multiple HTTP request to the server. This technique has already been used successfully to exploit IIS 5.0.

Stack. Relatively idle IIS processes copy the name of the requested resource (e.g. index.html) to a fixed offset on the stack of the thread servicing the request. The offset for the first thread is 0x9bf2cc.

Log. IIS uses memory-mapped IO to improve logging performance. A 64kB file block is allocated and mapped to memory. By default, the full query string is logged into this block, along with the server's response code. Whenever the 64kB of memory fill up, the data is written to disk and the write pointer is set to point to the beginning of the 64kB block. If we send enough repeated requests to IIS, we can fill the log with our url-encoded shellcode and have the shellcode at the beginning of the log with high probability. The base address of this memory block appears to vary within the range $0x3c0000 - 0x3d0000$ and can be guessed with several tries.

Hijacking Arbitrary Execution Control SCAs is not as reliable as Arbitrary Code Execution SCAs. Usually, mapping data to memory in this way has a non-zero probability of failure. Furthermore, selecting the proper approach requires a trial and error process, similar to SCA verification, that aims to check whether the hijacked exploit works.

Creating offline databases of memory invariants for multiple versions of software and multiple OSes is a time consuming task. However, we are constantly amazed at how subtle errors in code turn out to be exploitable in the hands of skilled attackers. Such a database can be constructed once, and then with infrequent updates, can be used for any new vulnerability that is later discovered. Thus this seems to be well within the capabilities of attackers. However the need to maintain different entries for different OSes and software versions may limit the reach of worms generated this way. This same limitation does not apply for targeted attacks (Section 3.2).

2.2.3 Arbitrary Function Argument SCAs

This case appears more difficult to automatically hijack in the general case. There are cases, however, that can be easily hijacked. For instance, if we control the parameters to the "exec" system function we can easily create a new exploit: previously fabricated shell scripts (that download the worm code and execute it) can be embedded in the message as parameters to the "exec" function. For other types of system calls, it is unclear how these can be used to automatically craft a new exploit.

Certain application level attacks can also be described using arbitrary function argument SCAs. SQL injection is an example, where the attacker partially controls the parameters passed to the SQL query engine: user-provided parameters used directly to construct SQL queries allow an attacker to execute SQL statements of its choice. Modern DBMS offer considerably more functionality than traditional DDL and DML statements, in some cases even allowing execution of arbitrary processes. An attacker can leverage this functionality to execute a command interpreter that downloads and executes the worm code. Candidates for SQL injection attacks are wide-spread open-source web software such as message boards, project management software, etc. The hijacker's database will include in this case the application name and the corresponding exploit code, along with a list of servers running this software. The list is trivial to create: popular search engines can be used to find pages with distinctive elements of the specific web application, such as logos, mottos, acknowledgements, etc.

2.3 Hijacking Using Network Detectors

Another approach to automatic worm containment is to combine network Intrusion Detection Systems (IDS) such as Bro [12] with automatic, network-based, mechanisms that generate worm signatures. These techniques use heuristics to first classify network flows as innocuous or suspicious and then search for recurring patterns within the suspicious flows [8, 10].

To hijack a worm using network based detectors, the attacker uses the output of the signature generation mechanisms to trace back into the message logs and identify the worm's attack packets. Executable code in these packets will be replaced with hijacker's own code.

However, the precision of network based signature generation is lower than that of host-based detectors, which have full access to host state. Consequently, hijacking using network based detectors will have smaller impact.

As with SCAs, if detectors alert a large number of network IDSes to stop the spread of the worm, the probability of a hijacker that owns such a node to discover an exploit early is increased, when compared to the case where the hijacker runs the network-based detectors himself.

3 Impact

So for we have concentrated on how to automatically hijack an exploit. Equally important from an impact point-of-view is how the hijacked exploit is then used, as this determines the possible defense strategies. We distinguish two uses of hijacking:

- *Auto-Worms*. Hijacking is used to create a worm that aims to outrun both the initial worm (if the original exploit was a worm) and SCAs generated to defend against the exploit.
- *Targeted Attacks*. The hijacker targets specific machines for infection. Software available on these machines is slowly mapped by the hijacker before the attack. When an exploit is detected, it is hijacked and immediately used to infect only these machines.

3.1 Auto-Worms

Botnets comprising desktop computers are comparatively easy to create (or indeed buy) using many different techniques such as email viruses and similar techniques. However, compromised *servers* have higher value in terms of the potential for malice or the economic damage that can be wreaked. Thus one motivation of an attacker is to leverage a cheap botnet into a much more valuable one. Alternatively an attacker simply wants to "own" more hosts.

In any case, the sooner the exploit is hijacked, the more machines are still unpatched (by the competing worm or SCA) to be subverted to the owner's control. To this end, the bot-net owner will both run his own detectors and register for the relevant SCAs. The more machines he uses for this, the higher the probability to discover the exploit early.

While passively waiting for an exploit to hijack, the botnet quietly creates hit-lists for the most popular software packages. When the exploit is hijacked by one of the bots, the resulting worm is rapidly disseminated to the other bots; each of these starts to infect its own portion of the hit-list, in an attempt to cut down the slow stage of the exponential spread and to compromise as many known hosts as possible before they are patched.

In light of this, an attacker discovering an exploitable vulnerability only has two choices: target very selectively so as not to trigger detectors, or create a really fast worm. Anything in the middle does not make sense, since someone else's auto-worm generated from his exploit will capture more vulnerable hosts. Similar competitive pressure is created by the SCA mechanism. This observation is particularly important: currently few worms are flash worms; it seems that pressure from both SCAs and hijacking obliges attackers to create flash worms.

Currently, few vulnerabilities are exploited by worms. Attackers seem to favor direct scanning from their bots as it is easier to avoid IDS systems. With hijacking it becomes much more likely that an exploit will become a worm. The ecosystem naturally pushes it that way, as direct scanning is likely to be too slow when competing with auto-worms.

Registering for SCAs highlights the opportunistic attitude of the hijacker. Although he will run detectors, these will commonly be on end-hosts that might not be early targets for server-based attacks, especially if his competitors are trying to avoid his detectors. In contrast, detectors for the SCA network are likely to be run on production servers to catch exploits early. Even if SCAs have propagated fast enough to protected most machines, the fraction the hijacker infects is still non-zero.

If not all vulnerable hosts register for SCAs, then the problem is significantly worse. In effect, if SCAs are used for a particular piece of software, it becomes necessary for *all* instances of that software to register to receive SCAs, or the risk is higher than if SCAs had not been deployed at all.

3.2 Targeted attacks

Suppose a malicious party wants to cause economic damage to a particular company (or even a country). For this, many compromised machines in that company may be needed. Hijacking provides a way to target them directly. The malicious user maps out the company carefully and slowly, and builds a catalog of all the software the company uses and the machines it runs on. When an exploit is detected that matches the software, it is turned into an exploit that is targeted at the company's machines.

Targetted attacks have two advantages from the point-of-view of the attacker.

First, they can be used by an attacker that does not possess a botnet. Such an attacker cannot afford to run a large number of detectors, but he can register for SCAs for a wide range of software used by his target. When an SCA arrives, it is then a race to see whether the hijacked exploit can be generated and delivered before the SCA is received by the target and a filter is generated. If the target fails to register for SCAs, then the attacker will always win.

Second, if an attacker does possess a botnet, then there is a much higher likelihood that he will receive the SCA before the target does. This tilts the balance in favour of the attacker. The SCA distribution mechanism needs to notify everyone worldwide, whereas the attacker can bring a large number of bots to bear on a single destination.

4 Evaluation

The success of exploit hijacking—measured as the percentage of the number of target machines infected using *auto-worms* or *targeted attacks*—is highly dependant on the properties of the initial exploit (assumed to be a worm) and the defense mechanism (SCA network). Here, we evaluate these dependencies to get a feeling of the parameter space.

4.1 Simulation Setup

We use a simple packet-level discrete event simulator to simulate an overlay network with 100.000 hosts taken from [4]. Out of these 1.000 are super peers organized in a secure overlay, with every super peer connected to approximately 50 other peers. The rest 99.000 hosts are susceptible for infection. Each of the end-hosts is connected to one super peer. Overlay delays are computed using a transit-stub

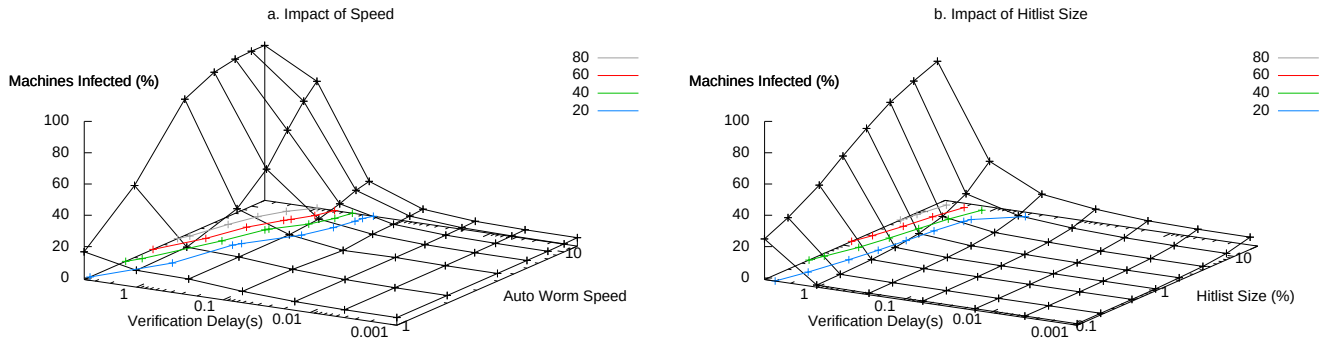

Figure 1. Auto Worm Impact

topology generated with the Georgia Tech Topology generator [16]. All the end-hosts are either vulnerable (i.e., such a host can be infected by the worm), detectors (that generate SCAs when hit by a worm) or bots (that hijack the worm or SCA it receives). Messages between hosts that are not directly connected in the overlay are assumed to have delay equal to the average delay of the network.

We use the infection model described in [7], modified to account for detectors and bots. Assume there are S susceptible hosts, with a fraction d of detectors and a fraction b of bots. Assume that the infection rates (also called worm speed throughout this paper) for the worm and auto-worm are β_w and β_a respectively. The equations describing the number of hosts infected by the worm (I_w) and the auto-worm (I_a) are:

$$\frac{dI_w(t)}{dt} = \beta_w I_t(1 - d - b - \frac{I_w(t) + I_a(t)}{S}) \quad (1)$$

$$\frac{dI_a(t)}{dt} = \beta_a I_a(1 - d - b - \frac{I_w(t) + I_a(t)}{S}) \quad (2)$$

Using the number of hosts infected by the worm or auto-worm and the equations above, we compute the time at which a host (selected randomly) will be probed by either the worm or the auto-worm. The bots are connected in a full mesh and have a hit-list of susceptible machines, which is selected as a fraction of the total vulnerable population. The worm is assumed to have a reference speed and does not use hit-lists. Whenever an exploit is hijacked by one bot, all the other bots are first notified and then each starts to infect its own share of the hit-list. Bots are assumed capable of sending 1.000 messages per second (to alert other bots or to infect the hosts from the hit-list). For simplicity, we assume that SCA and auto-worm generation are instantaneous, but account for the SCA verification lag, assumed to be the same for all hosts. The number of detectors is set to 1% of S, and the number of bots is 0.1% of S. Both are selected uniformly at random from the target population.

4.2 Results

First, we vary the speed of the auto-worm and the SCA verification time. The bots have a hit-list of 10% of the susceptible population. The results are presented in Figure 1.a. We see that even when SCA verification is really fast (1ms) and the auto-worm's speed is the same as the initial worm's, the auto-worm still infects 5% of the susceptible population. Increasing the speed of the worm brings benefits only when the SCA dissemination delay is higher. Otherwise, the auto-worm gets a fraction of the hit-list and few other hosts.

In reality, the SCA verification delay is expected to be high: in [4], the authors report SCA verification times on the order of milliseconds for verifiers that have an active running instance of the vulnerable software in a virtual machine when the SCA is received. If the software is started when the SCA arrives, verification takes a few seconds [4]. We expect similar delays for inactive processes (i.e., cold caches). When the SCA verification delay is on the order of seconds, the auto-worm infects a larger fraction of the population. Fast auto-worms (4 times as fast as the initial worm) infect approximately 20% of the population if the SCA verification delay is 1s and 80% if the delay is 4s.

The impact of the size of the hit-list on the number of hosts infected by the auto-worm is presented in Figure 1.b. The auto-worm is 4 times as fast as the initial worm. We see that the size of the hit-list matters greatly. If the auto-worm uses small hit-lists and SCA propagation times are small, the number of infected hosts is close to 0. If SCA propagation times are large (1s-4s), the increase is sharp when hit-list size increases.

Figure 2 presents the number of hosts infected by the worm and the auto-worm as a function of auto-worm relative speed (2.a, hit-list size 1%) and hit-list size (2.b, with the auto-worm as fast as the worm). We see that high speeds and hit-lists make the difference in the race between the two worms.

In Figure 3, we quantify the effectiveness of targeted

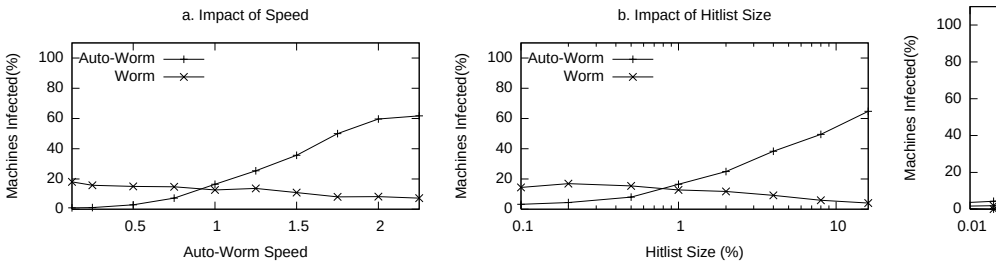

Figure 2. Worm vs Auto-Worm

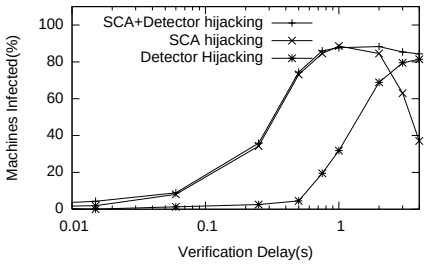

Figure 3. Targeted Attacks

attacks by measuring the percentage of infected machines out of the target group. We use 0.01% bots ($\sim$ 10) and select a target group uniformly containing 1000 machines. The initial worm's speed is set to be competitive with the SCA mechanism: when the SCA verification delay is 1s, the worm infects 17% of the population. We measure the fraction of target hosts infected by the bots as a function of SCA verification delay for three cases: when hijacking uses both SCAs and detectors, only SCAs and only detectors. When using both techniques, hijacking is most successful. Using only SCAs for hijacking (which is the case where detectors cannot be run, due to increased software diversity) brings benefits only when SCA verification is fast; otherwise, the worm will infect most of our targets. Using detectors only is effective when the SCA network is slow.

4.3 Discussion

The model we have used is simple and has a number of inaccuracies. First, all the hosts are considered vulnerable, which is the ideal case for the worm and auto-worm but also allows SCAs to be propagated by all hosts. Software diversity seriously complicates the SCA dissemination problem: few of the hosts will be able to forward any particular type of SCA. The problem is even worse for the super peer core: these must be able to forward all SCAs and therefore must run all possible versions of software.

Multi-hop routing is simulated by using average end-to-end delay, as opposed to using a shortest-path algorithm. Therefore, these results are expected averages; we cannot predict the extremes. This is particularly important for targeted attacks.

Worm outbreaks create significant traffic loads that cause packet losses. Here, we do not account for this type of behavior: we are more interested in the relative variation of the hijacker's success rate rather than the absolute value.

Finally, worm hijacking and SCA creation are assumed instantaneous. We expect hijacking to have similar overhead when compared to SCA creation, and therefore the comparison is fair.

We believe that all these drawbacks do not hinder the qualitative observations resulting from our experiments: worm speed and creating hit-lists matter greatly in online warfare: this pushes an attacker towards flash worms or towards niche attacks, where detectors are not present. SCA verification delay is equally important. If SCA verification delays are large, they will limit the effectiveness of SCA protection against worms and auto-worms. Finally, targeted attacks are relatively cheap to mount and quite successful when using both SCAs and detectors for propagation. Therefore, SCAs have an important and unwanted side-effect, allowing resource-scarce attackers to infect hosts of their choice.

5 Defenses

Automatic hijacking exploits is already feasible. Defense against such exploits can come in two ways:

- Design operating systems in such a way that automated worm generation will not work.
- Design alert mechanisms that can outrun the fastest worms and mitigate the effects of hijacking.

5.1 Operating System Design

One way to limit the effectiveness of *auto-worms* is to increase software diversity through randomness. Stack addresses can be randomized, by selecting a random base address and even selecting random addresses (with restrictions) for individual stack frames. This hinders attacks that use fixed addresses on the stack, but will not stop "jmp esp" attacks. Randomizing base addresses of DLLs and the code segment can remove the latter problem. Finally, randomizing the heap will make it much more difficult to predict addresses allocated on the heap, even for idle processes. These techniques must be combined with techniques for marking the stack and heap as non-executable. Such techniques are already being implemented into mainstream operating systems (e.g., Microsoft XP SP2 has built in Data Execution Protection). If these techniques are enabled, neither hijacking nor SCA-generation are possible for arbitrary code execution and arbitrary execution control attacks.

Randomization and non-executable pages are not able to stop arbitrary function argument attacks ("return into libc") or application level attacks. Therefore, they are not a complete solution for the hijacking problem.

5.2 Alert Mechanisms Design

Hijacking using detectors is feasible and can be exploited by culprits with enough machines. In this section we discuss how alert distribution mechanisms can be enhanced, in order to minimize their negative effects in terms of hijacking.

SCAs are an active alert mechanism. If an SCA can indeed outrun the fastest worms, then SCA hijacking into *auto-worms* does not greatly matter, as the newly generated worm cannot outrun the existing SCAs. Thus the performance of distribution networks for such alerts is critical.

To minimize the usefulness of SCAs for hijacking, we have two options: either stop forwarding SCAs to end-hosts and protect them using network level filters, or ensure that SCAs are received by all the hosts simultaneously and as fast as possible. Using these ideas as building blocks, we sketch two solutions and outline their pros and cons.

Large companies may be able to build distribution networks that validate and spread an SCA to a large number of their own servers before finally alerting the end recipients as simultaneously as possible. Of course if the original SCA was itself generated by detecting a fast worm, then it becomes of critical importance that the internal distribution network propagates SCAs as fast as possible. Any delay here results in *all* customers remaining unpatched while the worm runs unchecked. It is unclear what smaller companies and open-source software authors can do to compete. They cannot afford a special-purpose distribution network, so SCA propagation would need to be done via a peer-to-peer network. This is likely to be slower, as for robustness SCAs may need to be checked before forwarding. In addition, such a network is more susceptible to SCA hijacking.

Adding Trust to SCA Dissemination

We assume that border infrastructure servers are trusted. If these coincide with a host's ISP, the trust relationship is reasonable; the ISP can always cause denial of service to the host, so DoS using fake filters does not pose new threats. End-hosts will be provided with filters that drop worm attack messages but do not provide explicit information about the vulnerability. However, even filters can be used for hijacking: given a packet stopped by the filter, the hijacker can analyze it looking for executable code and paste its own code over it.

Going a step further, the ISP can install the filter locally without forwarding it to the host. Assuming the task of maintaining the personalized filters is feasible, this solution eliminates the need for end-hosts to receive SCAs.

However, if servers in the dissemination network are not trusted, we have the problem that all infrastructure servers must receive the SCAs in the same time. To this end, we can create a full mesh of nodes as the core SCA dissemination infrastructure and send each SCA to all the nodes simultaneously. Considering that SCAs are small ($<$1kB) and the number of servers is small (1000), the server has to send a total of 1MB of information to alert all servers. Since these nodes are likely to be well provisioned, this task can be achieved in less than a second. If IP multicast is deployed, the task is even simpler. This solution creates the potential for DoS attacks. To mitigate this, super-peers that send fake SCAs can be excluded from the network.

Controlling Information Leaked by SCAs

To counter the effects of hijacking, we can provide hosts with credible information that there exists an exploitable vulnerability, while in the same time avoiding to disclose complete information about it.

A simple idea is to masquerade one type of SCA as another type of SCA, e.g. arbitrary code execution can be transformed into arbitrary execution control, which are more difficult to hijack. This technique is unlikely to be effective, as it would require code-scrambling techniques that do no possess enough entropy to fool the hijacker.

A more promising approach is the following: whenever a new exploit is detected, the alert mechanism will first deliver a special type of warning to the vulnerable hosts. These must take preventive action until the exploit is confirmed by the SCA, by either stopping packets destined to the vulnerable service or running the vulnerable service in a virtual machine to minimize the eventual damage. The SCA will be delivered after enough time has elapsed to ensure that the wide majority of hosts have received the warning. Upon receipt of the SCAs, the hosts will either create filters if the SCA is valid, or take some punitive action against the detector if the SCA is fake.

We propose two types of warnings. The first is inspired by a category of cryptographic protocols termed Zero Knowledge Proofs (ZKP, see definition by Goldreich [6]). We can apply ZKP protocols to SCAs as follows: the detector can prove using ZKP protocols to the vulnerable end-host or forwarder that a piece of software is vulnerable (which is NP to determine in the general case) without disclosing more information about the exploit than necessary. However, ZKP protocols involve multiple-rounds and are therefore time consuming, being too expensive for our setting. The practical version we propose is to masquerade all SCAs as packets that cause the vulnerable service to crash. Whenever a host receives an SCA that crashes its software, it can infer that there is a non-negligible probability that the bug is exploitable and can take preventive action. Creating such warnings can be easily achieved by having the detector insert random entries into the the payload, by overwriting either the jump addresses (for arbitrary execution control vulnerabilities) or instruction opcodes (for arbitrary code execution vulnerabilities).

The second type of warning uses commitments: these allow a sender (the detector) to commit to some data (the SCA) and send the commitment to the receiver (the vulner-

able end-hosts or forwarders) without disclosing the details of the SCA. After some time has elapsed, the sender reveals the secret to the receiver. A correct commitment scheme ensures that the sender cannot claim to have committed to another value. The danger with this scheme is that it creates the opportunity for DoS attacks, since anybody can create such warnings. However, the originator of the SCA can be held accountable for its contents, and therefore malicious detectors can be excluded from the network.

6 Summary

In this paper, we have outlined an important side-effect of automated exploit defenses: *hijacking*. This allows an attacker to transform an existing exploit into a worm or exploit that works in its benefit. This is worthwhile from the point of view of the hijacker, which does not need to undertake the difficult task of finding and exploiting a vulnerability. The hijacker can prepare while waiting for somebody else to discover an exploit, and hijack it either to target directly a group of machines or to infect a fraction of the vulnerable hosts with an *auto-worm*. Hijacking changes the threat landscape: an attacker that has an exploit can only target very selectively or create a flash worm; any attack between these two extremes will not survive.

We have provided evidence that hijacking is indeed possible, not only for resource-rich hijackers that are able to run detectors, but also for small scale hijackers that leverage Self Certifying Alerts. There is a tight relationship between what can be described accurately using an SCA and what can be hijacked.

Through simulation, we have explored the problem space showing that if the hijacked worm is fast and/or uses hitlists, it outruns the initial worm to a larger fraction of hosts. Results show that such an auto-worm is competitive with the SCA dissemination mechanism when verification delays are on the order of seconds.

Finally, we have presented possible defenses against hijacking, spanning from operating system design to alert mechanisms design, that appear applicable and could be used in this setting. However, bundling these initial attempts into a complete solution is challenging. We leave as an important open problem to devise efficient worm defense techniques that are resilient to hijacking.

7 Acknowledgement

We would like to thank Jon Crowcroft and Manuel Costa for numerous insightful discussion on this topic and for providing us with simulation tools and data. Costin Raiciu is supported by a UCL Departmental Studentship. David Rosenblum and Mark Handley hold Wolfson Research Merit Awards from the Royal Society.

References

[1] Findjmp2. `http://www.derkeiler.com/Mailing-Lists/Securiteam/2005-02/0067.html`.

[2] Generic connectback shellcode for win32. `http://www.hick.org/code/skape/shellcode/win32/connectback.c`.

[3] F. Castaneda, E. C. Sezer, and J. Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 83–93, New York, NY, USA, 2004. ACM Press.

[4] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[6] O. Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.

[7] H. W. Hethcote. The mathematics of infectious diseases. *SIAM Rev.*, 42(4):599–653, 2000.

[8] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[9] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[10] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[11] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[12] V. Paxson. Bro: a system for detecting network intruders in real-time. *Comput. Networks*, 31(23-24):2435–2463, 1999.

[13] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[14] A. Smirnov and T. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.

[15] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[16] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM*, pages 594–602, 1996.