Exploit Hijacking: Side Effects of Smart Defenses

Costin Raiciu
Department of Computer Science
University College London
c.raiciu@cs.ucl.ac.uk

Mark Handley
Department of Computer Science
University College London
m.handley@cs.ucl.ac.uk

David S. Rosenblum
Department of Computer Science
University College London
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 extremes (targeting selectively or creating a rapidly spreading worm), and 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 by Costa et al. for end-to-end worm containment. We provide a discussion of the effects of hijacking on the threat landscape and list a series of possible countermeasures.

Keywords

exploit hijacking, self-certifying alerts

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 [8, 13, 16, 20, 22]. 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 already has 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 detailed information about the exploit. In many cases this is enough to generate *automatically* 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 [24]. 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 is Self-Certifying Alerts [7]. 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 generate a filter automatically 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.

SIGCOMM'06 Workshops September 11-15, 2006, Pisa, Italy.

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 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 Detectors

Instrumented software designed to detect attempted breakins works by tracking 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 [8,16,20,22]. By tracing back the tainted data to its origin in the message logs, the detector finds the message that contained the exploit. Normally this process 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 antiworms) was proposed by Castaneda et al. [5]. The techniques described 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, by replaying the exploit in a sandboxed version of the vulnerable application. Three types of SCA are detailed by Costa et al. [7]:

- *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 Costa et al. 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 success 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.

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, ...

Figure 1: Arbitrary Execution Control SCA for Slammer [7]

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 and 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 our first attempt, without any debugging, for both worms.

2.2.2 Arbitrary Execution Control SCAs

Leveraging Arbitrary Execution Control SCAs is a bit more complicated than Arbitrary 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 is 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, both of which use stack-based buffer overflows, can 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 and is aimed for exploits that are not stack overflows. Creating a database of memory invariants in the target software is not an easy task, but it is often feasible.

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. 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 requests to the server. This technique already has 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 I/O 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 a new file block is mapped at the same memory location. If we send enough repeated requests to IIS, we can fill the memory mapped block with our URL-encoded shellcode and have the shellcode at the beginning of the log with high probability. The base address of the 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, to check whether the hijacked exploit works.

Creating offline databases of memory invariants for multiple versions of software and operating systems is time-consuming. 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 discovered later. Thus, this seems to be well within the capabilities of attackers. Nevertheless, the need to maintain different entries for different operating systems 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*

These SCAs appear to be the most difficult to hijack automatically in the general case. There are cases, however, when they can be hijacked easily. For instance, if we control the parameters to the "exec" syscall, we can easily create a new exploit: Previously fabricated shell scripts (that download the worm code and execute it) can be provided as parameters to "exec". For other types of system calls, it is unclear how these can be used to craft a new exploit automatically.

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 his choice. Modern Database Management Systems (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 widespread 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, acknowledgments, etc.

2.3 Hijacking Using Network Detectors

As an aside, we briefly speculate how network-level detectors can be used for hijacking. Network Intrusion Detection Systems (IDS) (such as Bro [17]) combined with automatic network-based mechanisms that generate worm signatures can automatically stop spreading worms. The latter techniques use heuristics first to classify network flows as innocuous or suspicious and then to search for recurring patterns within the suspicious flows [12, 15].

To hijack a worm using network detectors, the attacker uses the output of the signature generation mechanism to trace back into the message logs and identify the worm's attack packets. Executable code in these packets will be replaced with the hijacker's own code. In some cases where the code possesses a certain structure that precludes simple overwriting, state machines or even host-based detectors must be used to guide hijacking. If the code is encrypted, hijacking is not possible at network level.

In general, the precision of network-based signature generation seems lower than that of host-based detectors, which have full access to host state. Consequently, we expect hijacking using network detectors will have smaller impact.

3. IMPACT

So for we have concentrated on how to hijack an exploit automatically. 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 mapped slowly 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. 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 botnet owner will run both his own detectors and register for 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, but only if SCAs are *deployed globally*. However, global deployment of SCAs, although highly necessary, does not appear to be imminent; hijacking, on the other hand, is already feasible.

This observation is particularly important: Currently, few worms are flash worms; it seems that pressure from both hijacking and SCAs obliges attackers to create flash worms. Also, few vulnerabilities are exploited by worms. Attackers seem to favor direct scanning from their bots as it avoids IDS systems more easily. 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 protect 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 for 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.

Targeted 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 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 favor 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.

A special type of targeted attack is an attacker controlling an ISP node (or equivalent). In this case, existence of SCAs implies that this node can infect all the machines in its subnet; since it controls traffic to that sub-net, it will drop SCAs and will use them only for hijacking.

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 dependent on the properties of the initial exploit (assumed here to be a worm) and the defense mechanism (SCA network). Here, we evaluate these dependencies through simulation.

4.1 Simulation Setup

We use a simple packet-level discrete event simulator to simulate an overlay network with 100,000 hosts similar to that used by Costa et al. [7]. Of these, 1,000 are super peers organized in a secure overlay, each connected to about 50 other peers. The remaining 99,000 hosts are susceptible to infection. Each end-host is connected to one super peer. Overlay delays are computed using a transit-stub topology generated with the Georgia Tech topology generator [26]. Each end-host is either vulnerable (i.e., such a host can be infected by the worm), a detector (that generates SCAs when hit by a worm) or a bot (that hijacks 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 by Hethcote [10], 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:

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

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

Using the number of hosts infected by the worm or auto-worm and the equations above, we compute the time at which a randomly selected host will be probed either by the worm or the auto-worm. The bots are connected in a full mesh and have a hit-list of susceptible machines, selected as a fraction of the total vulnerable population. Each infected host probes a random host every 10ms. The worm does not use hit-lists. Whenever an exploit is hijacked by one bot, all other bots are 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

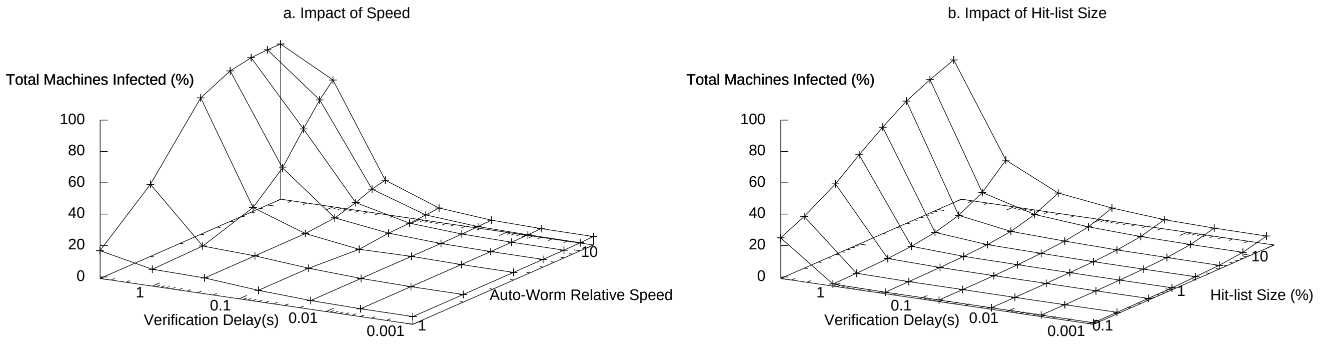

Figure 2: Auto Worm Impact

the hosts from the hit-list). For simplicity, we assume that SCA and auto-worm generation are instantaneous, but we do model the SCA verification lag, which is assumed to be the same for all hosts. To make the analysis manageable, we arbitrarily chose a random 1% of S to be detectors, and 0.1% to be bots. These values seem reasonable, but as the results do depend on this ratio, they should only be read as illustrative.

4.2 Results

First, we investigate how the speed of the auto-worm and the SCA verification time affect the outcome. To do this, we fix the size of the bots' hit-list to be 10% of the susceptible population. Figure 2a shows that even when SCA verification is very fast (1ms) and the auto-worm's speed is the same as the initial worm's, the auto-worm still infects 5% of the hosts. Increasing the speed of the auto-worm only improves its success when the SCA dissemination delay is fairly high. Otherwise, the auto-worm only infects a fraction of its hit-list and few other hosts.

In reality, SCA verification may be quite slow: In Costa et al. [7], SCA verification times of the order of milliseconds are reported for verifiers having an active running instance of the vulnerable software in a virtual machine when the SCA arrives. However, if the software is started when the SCA arrives, verification takes a few seconds [7]. We expect similar delays for inactive processes (i.e., cold caches or paged out). When SCA verification takes 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 shown in Figure 2b. In this figure, the auto-worm was four 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 zero. If SCA propagation times are large (1s–4s), the increase is sharp when hit-list size increases.

Figure 3 presents the number of hosts infected by the worm and the auto-worm as a function of auto-worm relative speed (Figure 3a, hit-list size 1%) and hit-list size (Figure 3b, with the auto-worm as fast as the worm). We see that high speeds and large hit-lists make the difference in the race between the two worms.

In Figure 4, we quantify the effectiveness of *targeted* attacks by measuring the percentage of machines from the target group that become infected. As this scenario is one that a resource-poor attacker can perpetrate, we use only 0.01% bots (approximately 10). The target group contains 1000 randomly chosen 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, uses only SCAs, and uses 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) is beneficial only when the SCA dissemination is fast. Otherwise, if SCAs are slow to reach the hijacker, the initial worm gets the largest fractions of the vulnerable hosts (Figure 4, when verification delay is greater than 1s). In this case, using detectors makes hijacking more effective.

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, causing packet losses(e.g., see [14]. 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 the cost of hijacking to be similar to that of SCA creation, and so the comparison is fair.

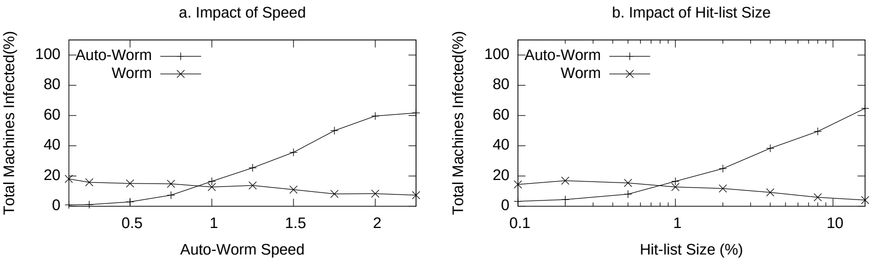

Figure 3: Worm vs Auto-Worm

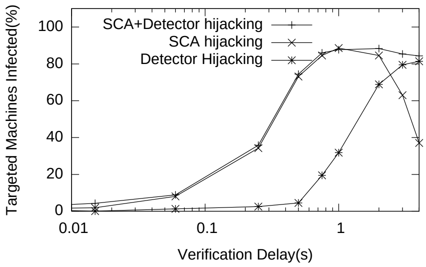

Figure 4. Targeted Attacks

We believe that 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. Finally, targeted attacks are relatively cheap to mount and quite successful when using both detectors and SCAs for hijacking. Therefore, SCAs have an important and unwanted side effect, helping resource-poor attackers to infect hosts of their choice.

5. DEFENSES

Automatic hijacking of exploits is already feasible; steps must be taken to defend against this type of threats. Since the only difference between auto-worms and targeted attacks is the shellcode in the attack packets, we consider defenses that tackle both types of attacks.

Defenses for hijacking must be built on top of current defenses against exploits, which fall into two broad categories:

- Decrease software homogeneity with randomness.
- Use software homogeneity to cooperatively monitor and guard against new vulnerabilities.

We now discuss each technique separately.

5.1 Reducing Software Homogeneity

One well-known way to limit the effectiveness of exploits is to increase software diversity through randomness [3, 6, 25]. Techniques in this class do not need to be changed to protect against hijacking. We describe here some of the main results only to point out that the second class of defenses—exploiting homogeneity—is necessary.

Address space randomization works by selecting random base addresses for the stack, heap and code segments at compile-time, link-time or even run-time [3,6,25]. Randomizing base addresses of the code segment and even the ordering of functions within the code segment is used to avoid "jmp esp"-type attacks by making the location of any code containing "jmp esp" unpredictable. Randomizing the heap makes it much more difficult to predict addresses allocated on the heap, even for idle processes. Randomizing syscall numbers can also be helpful against arbitrary function argument attacks.

Certain application level attacks can be countered by using context-sensitive string evaluation [18]. SQL injection can be mitigated with SQL randomization techniques [4].

Instruction Set Randomization adds entropy to the instruction set, making code-injection attacks difficult [11].

These techniques are combined with techniques for marking the stack and heap as non-executable, which are already being implemented in mainstream operating systems (OpenBSD's W⊕X, Microsoft XP SP2's Data Execution Protection). If these techniques are enabled, neither hijacking nor SCA-generation are possible for arbitrary code execution attacks.

Unfortunately, randomization and W⊕X techniques are not a panacea. It has been shown that the amount of entropy available on current 32-bit architectures (16-20 bits) for address-space randomization does not offer protection against a brute force attack [19], and that stack-overflow attacks can be exploited even if W⊕X is enabled, by using a variant of the "return to libc" attack [19]. Instruction set randomization can also be broken in some cases, by using an attack that incrementally discovers the key used for randomization [23].

5.2 Using Software Homogeneity

Harnessing software homogeneity, it is possible to safeguard members of an application community using distributed detection and fast dissemination of vulnerability information [7, 21]. Clearly, existing taint-tracking detectors can be used for hijacking, increasing the aggressiveness of exploits even in the absence of SCAs. We believe that active alert mechanisms such as SCAs are highly needed, given that techniques based on randomization have limited effectiveness. 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 SCAs 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 SCAs is critical. The effects of SCA hijacking can be minimized by ensuring that all hosts receive the SCA as simultaneously as possible.

In the current peer-to-peer model [7], SCA dissemination must trade-off the need to alert everybody (almost) simultaneously against vulnerability to denial of service attacks. Since hosts or detectors are not trusted, SCA propagation is pruned by the verification process, such that fake SCAs are dropped early to minimize DoS effects. The effectiveness of pruning depends on the maximum number of overlay hops the SCA travels, being more effective as the number of hops increases. However, the danger of hijacking is minimized if all the hosts receive the SCA within one round trip time (by using IP multicast, for instance), in a single overlay hop. It is unclear what the proper trade-off is in this model, or if such a trade-off even exists.

Large software vendors may be able to build special-purpose distribution networks that validate and spread an SCA to a large number of their own servers before finally alerting end recipients as simultaneously as possible, perhaps using auxiliary distribution channels such as the broadcast TV network. In this paper we do not explore this possibility, focusing instead on ways to enhance the current peer-to-peer model. While large companies may be able to use other models, small companies and open-source are probably constrained by economics to peer-to-peer style solutions.

Our proposed solution is to add a degree of trust to the SCA dissemination infrastructure, and it has two variants that can be used in tandem. The first proposal is not to forward SCAs to end-hosts, but to protect them using network level filters (therefore trusting ISPs). The second proposal is to trust detector nodes to some extent, and "simulate" synchronous SCA distribution.

Limited Dissemination of SCAs

One option is to send SCAs only to ISPs. Thus, a smaller number of (hopefully more trusted) machines will receive the SCA. Each ISP will then apply measures to protect its customers against spreading worms.

A naive approach would be to have ISPs provide end-hosts with filters that drop worm attack messages when SCAs are received and verified. Although such filters do not provide explicit information about the vulnerability, they might still be used for hijacking. Thus we need a better alternative.

We slightly modify the previous approach: The ISP installs the filter locally, instead of forwarding it to the host. If the task of maintaining personalized filters for a large number of customers is feasible, this solution eliminates the need for end-hosts to receive SCAs. It implies that the end-hosts trust the ISP not to drop their packets arbitrarily; however, ISPs can do this even in the absence of SCAs. A drawback of this approach is that it does not protect small networks (i.e., LANs, community networks, etc.), only limiting global worm propagation. Considering that other infection vectors such as email viruses are available for a worm, this flaw is worth considering; it can be mitigated by applying in small scale networks the two-phase dissemination scheme we describe next.

Two-Phase Dissemination

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

One simple idea is to disguise one type of SCA as another type of SCA which is more difficult to hijack (e.g. arbitrary code execution can be transformed into arbitrary execution control). This technique is unlikely to be effective, as it would require code-scrambling techniques that do not possess enough entropy to fool the hijacker.

An alternative approach is the following: whenever a new exploit is detected, the first phase of alert delivers a preliminary warning to all the vulnerable hosts. These hosts must take preventive action until the exploit is confirmed by the SCA, such as pausing traffic to the vulnerable service. The second phase actually delivers the SCA, but only after enough time has elapsed to ensure that the vast 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 preliminary warning was fake. In effect, this simulates simultaneous delivery of SCAs.

We propose two types of warnings. The first is inspired by a category of cryptographic protocols termed Zero Knowledge Proofs (ZKPs, see definition by Goldreich [9]). Using ZKP protocols, the SCA detector can prove to the vulnerable end-host or forwarder that a piece of software is vulnerable (which is NP-hard 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 for the preliminary warning to be a modified SCA which would not exploit the service, but rather cause a crash. Whenever a host receives a warning that causes a crash in its SCA validator, it can infer that there is a non-negligible probability that the bug is exploitable and can take preventative action. Creating such warnings can be done by having the detector insert random entries into the the payload, by overwriting either the jump addresses (arbitrary execution control vulnerabilities) or instruction opcodes (arbitrary code execution vulnerabilities). This type of warning only adds a small amount of trust in detectors (i.e., that the software flaw advertised by the SCA is exploitable) but it has a side-effect, allowing a culprit that also uses *detectors* to use the speed of the SCA network to find the exploit as soon as possible.

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 vulnerable 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. This is reasonable if the number of detectors is relatively small and they are "known" by the core dissemination infrastructure (i.e., with PKI).

6. SUMMARY

In this paper, we have outlined an important side effect of automated exploit defenses, *hijacking*, which allows an attacker to transform an existing exploit into a worm or exploit that works to the benefit of the attacker. This is worthwhile from the point of view of the hijacker, who need not 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 either target very selectively or create a flash worm. However, 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 appears to be a tight relationship between what can be described accurately using an SCA and what can be hijacked.

We have explored the ensuing race through simulation, showing that if the hijacked worm is fast or uses hit-lists, 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 listed possible defenses against hijacking, ranging from operating system design to alert mechanism design. These defenses appear viable, but bundling these initial attempts into a complete solution is challenging. We believe that devising efficient worm defense techniques that are resilient to hijacking remains an important open problem.

7. ACKNOWLEDGMENTS

We would like to thank Jon Crowcroft and Manuel Costa for numerous insightful discussions on this topic and for providing us with simulation tools and data; Andrea Bittau and Brad Karp for discussions and reviews on previous versions of this paper; and the anonymous referees for their detailed feedback. Costin Raiciu is supported by a UCL Departmental Studentship. Mark Handley and David Rosenblum hold Wolfson Research Merit Awards from the Royal Society.

8. 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] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[4] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *ACNS*, 2004.

[5] F. Castaneda, E. C. Sezer, and J. Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *ACM workshop on Rapid malcode(WORM)*, 2004.

[6] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.

[7] 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*, 2005.

[8] 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), 2002.

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

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

[11] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, Oct 2003.

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

[13] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[14] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. Technical report, CAIDA/SDSC/UCSD, 2003.

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

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

[17] V. Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Networks*, 31(23-24), 1999.

[18] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Lecture Notes in Computer Science*, volume 3858, Oct 2006.

[19] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

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

[21] S. Sidiroglou, M. E. Locasto, and A. D. Keromytis. Software self-healing using collaborative application. In *NDSS*, 2006.

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

[23] N. Sovarel and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *USENIX Security*, 2005.

[24] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *USENIX Security Symposium*, 2002.

[25] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.

[26] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM*, 1996.