String Oriented Programming

Circumventing ASLR, DEP, and Other Guards

Mathias Payer

mathias.payer@nebelwelt.net

ETH Zurich, Switzerland

Abstract

The protection landscape is changing and exploits are getting more and more sophisticated. Exploit generation toolkits can be used to construct exploits for specific applications using well-defined algorithms. We present such an algorithm for leveraging format strings and introduce string oriented programming.

String oriented programming takes format string exploits to the next level and turns an intrusion vector that needs hand-crafted exploits into arbitrary code execution. Similar to return oriented programming or jump oriented programming sting oriented programming does not rely on existing code but concatenates gadgets in the application using static program analysis.

This paper presents an algorithm and a technique that takes a vulnerable application that contains a format string exploit as a parameter and constructs a format string exploit that can be used to inject a dynamic jump oriented programming dispatcher into the running application. String oriented programming circumvents ASLR, DEP, and ProPolice.

1. Introduction

Smashing the stack or the heap are two well known techniques to exploit running applications that have been around for quite some time. Many compilers and kernels protect from these basic forms of attacks using write $\oplus$ execute pages, mapping the stack non-executable, adding canaries on the stack, or using other techniques. All the protection that is nowadays in place renders these code injection attacks (almost) impossible.

Return to libc and more general return oriented programming (ROP), pointer subterfuge, and jump oriented programming (JOP) are two modern techniques to circumvent current protection mechanisms. These forms of attacks no longer rely on injected executable code but reuse the available code of the application for their malicious intent.

ROP relies on an unchecked application stack, i.e., return addresses on the stack are not verified. Modern runtime guards use a separate shadow stack to disable ROP based attacks (with a small runtime overhead). JOP based attacks are more complicated to run and also more complicated to protect against. A runtime system either checks the integrity of every dynamic control flow instruction or the compiler ensures that no open dynamic control flow instructions (e.g., `jmp *%eax`; an indirect jump through the `eax` register) are available in the compiled source. These attacks are all well understood and several protection schemes already exist to protect against classes of bugs and other protection schemes are emerging to protect against the remaining ones.

One different class of bugs has not yet received considerable attention under the new non-executable, no-code-injectable policy, namely format string vulnerabilities. If a user-controlled string is passed as a first argument to a function of the `printf` family then the string is parsed as a format string. Using such a bug and special format markers can be used to execute arbitrary memory reads on the stack and arbitrary memory writes to any location. Existing exploits use format string vulnerabilities to mount stack or heap-based code injection or to set up return oriented programming.

This paper takes a different approach and uses format string vulnerabilities from a different angle. We describe an algorithm to exploit arbitrary binaries. Parameters to the algorithm are the vulnerable application, the location of the format string vulnerability in the current shared object, the code sequence that should be executed, and a set of restrictions for the input string.

This paper focuses on x86 and a Unix-like operating system (e.g., Linux). The concepts apply to other platforms and operating systems as well.

2. Attack model

This paper assumes the following attack model. An attacker with restricted privileges tries to escalate privileges using an exploit. The attacker is either remote and tries to get user access or the attacker is local and tries to attack a "SUID" based binary to get administrator privileges.

This section describes existing attack vectors that are used to exploit an application. A successful attack redirects the control flow of the application to an alternate location (i.e., new code is injected into the application) or executes already existing code in a different context (i.e., existing code is executed with different, i.e., malicious, data). Both forms of attack rely on the following features:

1. The runtime environment must allow the redirection of the control flow to alternate locations using a control flow transfer instruction[1]. Indirect control flow transfers (indirect jumps, indirect calls, and return instructions) read

[1] Control flow transfer instructions are jump instructions, indirect jump instructions, conditional jump instructions, call instructions, indirect call instructions, interrupts, and system calls. For all instructions except indirect jump instructions, indirect call instructions, and return instructions the target is at a fixed address which is relative to the current location or at an absolute location in memory. These instructions encode the offset directly in the instruction itself. The target of these instructions cannot be changed if the code region is not writeable and are not useful for exploits.

the absolute target address from a data region. Exploits overwrite such a data pointer (e.g., EIP on the stack, a function pointer on the heap, a GOT entry in a shared library, vtable entries of objects, or data-structures of the memory allocator [6]) for the initial control flow redirection.

2. The exploit must inject some form of payload into the application. Code-based attacks inject machine code instructions into an executable region in the memory space of the application. These instructions are executed after the initial control flow redirection. Data-based attacks modify data structures of the application, a shared library, or the standard loader to execute their malicious payload.

An exploit is only successful if both requirements are met. The following sections present the four possible attack vectors in more detail.

2.1 Code injection

A code injection attack writes additional code into an executable region of the application's memory image and transfers control to that injected code [2]. Code injection attacks often use a buffer overflow (e.g., for a C based string or array) to inject the code and to overwrite a stored instruction pointer in one step. Listing 1 shows a vulnerable C snippet that is prone to a stack-based buffer overflow. An attacker can inject any data into the buffer and write over the bounds of the buffer to overwrite data structures that are higher up in the stack frame.

```
int is_foobar(char *cmp) {
  // assert(strlen(cmp) < MAX_LEN)
  char tmp[MAX_LEN];
  strcpy(tmp, cmp); // no bound check
  return strcmp(tmp, "foobar");
}
...
// user_str is > MAX_LEN
if (is_foobar(user_str))
...
```

Listing 1. A potential stack-based overflow.

Listing 2 shows a heap-based data structure that is also vulnerable to a buffer overflow. The data structure contains a function pointer that is used to work with the data in the data buffer. This function pointer can be overwritten through a buffer overflow.

```
typedef struct vuln_struct {
  char buf[MAX_LEN];
  int (*cmp)(char*);
};
```

Listing 2. A struct that is vulnerable to a heap-based overflow.

Code injection is a very old attack vector and has been used for many years. Until recently Intel IA32 did not support the separation of code and data. The CPU tried to interpret any memory region as executable, enabling code injection attacks into data regions. The 64 bit extension x64 enables separation of data and code. Only code on pages that have the executable flag set is executed by the CPU. If an exploit redirects control flow to a data page then an exception is triggered. Nowadays most systems support $W \oplus X$; a memory page is either writeable or executable. Due to $W \oplus X$ this attack vector is only applicable under special circumstances (e.g., executable trampolines on the stack, shared memory regions with wrong permissions, exploitable just-in-time compilers).

2.2 Return oriented programming

Return oriented programming [15, 19, 21] (ROP) relies on a stack-based buffer overflow. A return oriented attack constructs a set of stack invocation frames that are executed one after the other. Each stack invocation frame prepares a set of parameters on the stack and targets a gadget[2] that uses the parameters and executes some computation.

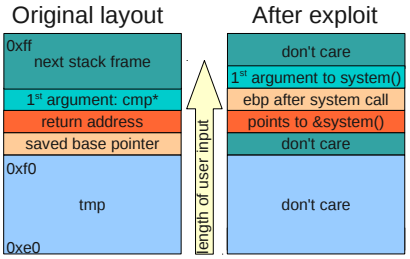

Figure 1. Stack before and after a stack-based arc injection.

Figure 1 shows a simple returned oriented programming attack that uses a buffer overflow to create one stack invocation frame that executes the `system()` function with forged parameters.

2.3 Jump oriented programming

Jump oriented programming [7, 19] (JOP) is similar to ROP. Both programming styles inject data to modify the control flow of the application using gadgets. Jump oriented data is not limited to stack overflows but uses modified indirect control flow transfers to construct the chain of executed gadgets. Indirect control flow transfers are used in the application to support, e.g., library calls, function pointers (callbacks), and object oriented programming.

2.4 Format string attacks

A format string attack [13, 16, 20] exploits an user-controlled string that is passed to a function of the `printf`-family. The `printf`-family parses the first string argument for control tokens (of the form `%T`) to determine the number of variable parameters that follows. Many programmers forget to check user-controlled strings for these control tokens and pass the string directly to the function (e.g., `printf(usr_str)`). A safe implementation would use a static parameter to pass a single string (e.g., `printf("%s", usr_str)`).

The `%n` token is a special control token that reverses the order of input. All other tokens specify a format that is used to print the current argument. `%n` writes the amount

[2] A gadget is a code snippet (not necessarily a function) that already exists in the memory image of the application.

of already printed characters to the address given by the current argument. Any argument on the stack can be used as a target address for `%n` with careful encoding of the format string. The format string itself can be used to store pointers to specific addresses if it is placed on the stack. The amount of written bytes can be controlled with additional parameters (e.g., `printf("%NNc");` prints NN bytes) and increases the counter used for `%n`.

The malicious format string can use other tokens like `%p` to read specific pointers on the stack, and `%s` to read specific stack addresses as strings. An attacker uses these parameters during the construction of the format string.

The combination of input tokens (e.g., `%p`, `%s`, and `%x`) are used to gather information about the stack layout (if the binary is not available). The gathered information about the stack frame is then used to construct an exploit string that consists of `%NNc` to set single byte values (i.e., to increase the number of written bytes) and `%hhn` to write single bytes to given memory addresses. The exploit string then writes arbitrary values to arbitrary memory locations. These random writes are then used to redirect control flow to injected code. The injected code is often in the string itself.

3. Protection mechanisms

Several protection mechanisms have been proposed that protect against one or more of the attack vectors shown in Section 2. The presented protection mechanisms try to detect possible attacks on different levels of granularity. The protection mechanisms either (i) check the integrity of the stack, (ii) verify library usage, (iii) encrypt pointers, (iv) change the instruction set, (v) protect format strings, (vi) randomize memory locations, or (vii) check and verify every single instruction that changes control flow.

The data layout for most languages places buffers and variables alongside with return instruction pointers and frame pointers on the regular application stack. Several protection mechanisms [10, 14] verify the stored return instruction pointer on the stack before the return instruction dereferences the stored address. These mechanisms protect from malicious changes of the return address and the stack layout.

Libsafe and Libverify [3] implement wrappers for library functions that are used in attacks. This approach protects from common errors and adds extra checks to "dangerous" functions. A disadvantage of this approach is that it only protects specific functions and general patterns of attack vectors. The glibc has a set of similar patches that are enabled if the `FORTIFY_SOURCE` is enabled. These patches check every parameter of format strings. The fortify patches are not secure and can be disabled at runtime [20].

Pointer encryption [9] is an interesting approach to protect instruction pointers from malicious changes. All instruction pointers are encrypted (e.g., using a hash). The application uses the encrypted pointers in all computation (e.g., comparing different function pointers). The compiler adds additional code that resolves the original instruction pointer using the given encrypted pointer whenever it is dereferenced. The attacker does not know the encryption function and therefore cannot forge a pointer to an arbitrary address. Instruction set randomization [12] is a similar approach. The application uses a randomized instruction set and an attacker is unable to guess the instruction set.

Format Guard [8] warns if format strings and functions of the `printf` family are used with unchecked user input. These guards protect the already existing functions of the libc but do not protect from format string exploits in the application code.

Address Space Layout Randomization (ASLR) [4, 5, 17] randomizes all dynamic data of an application (e.g., dynamically loaded libraries, heap, and stack). A potential exploit can no longer rely on hardcoded addresses for, e.g., library routines and gadgets. A drawback of this approach is that the address space is small and only a few bits are randomized. The limited randomization opens the possibility for probabilistic attacks [22].

Control Flow Integrity [1, 11] uses static binary translation to verify every target of all control flow transfers. A set of targets is associated with every control flow transfer location. The group of targets is identified with a secret number. This number is verified when the control flow transfer is executed. The control flow transfer is allowed only if the target number (in the target code) matches the verification code at the source location.

Libdetox [18] is a dynamic binary translation approach that uses runtime information to construct an control flow graph. This control flow graph is enforced at runtime using dynamic checks that are encoded into the translated code.

4. String oriented programming

String oriented programming (SOP) defines an algorithm that uses a format string based exploit and a set of constraints to execute arbitrary code of an adversary inside an application. SOP uses gadgets that are already available in the code region of the application. The algorithm assumes that some form of DEP[3] is enforced by the system. Otherwise it would be simple to inject some code and to redirect the control flow to the injected code.

Format string exploits enable writes with random values (encoded in the format string) to any memory location that is referenced through a pointer on the stack. An attacker can place arbitrary pointers in an attacker-controlled buffer on the stack; these pointers are then used in the format string attack to write arbitrary memory locations. We assume that the `FORTIFY_SOURCE` patches of the glibc are not enabled (or that we can call `printf` directly. The fortify patches can be disabled using the format string attack itself. The fortify patches only add some complexity (and length to the format string), see [20].

[3] Data Execution Prevention (DEP) uses the executable bit for pages in modern memory management units to enable non-executable data regions. DEP ensures that only code pages are executable. A stronger guarantee is $W \oplus X$ which ensures that a page is either writeable or executable but not both. Linux uses an $W \oplus X$ approach called *Exec Shield* [23] for the initial memory layout when applications are started.

4.1 Executing code

SOP uses two scenarios to get control of the application without executing injected code. The scenarios are similar to either ROP or JOP.

Scenario 1 uses a random write and a user-controlled buffer on the stack (often the format string itself) to prepare the attack. The random write redirects control flow to a gadget that adjusts the stack frame to the attacker-controlled buffer. The buffer contains a set of invocation frames that concatenate several available gadgets to execute arbitrary code. This approach combines format string exploits and ROP similar to Section 7.2 of [15].

Scenario 2 uses a random write and a user-controlled buffer on the heap (at a fixed address) to prepare the attack. The random write is used to redirect control flow to a first gadget that prepares the dispatcher for JOP. The attacker-controlled buffer can be used to store additional data for the JOP dispatcher (e.g., the chain of gadgets).

4.2 Resolving addresses

Without ASLR, SOP can be used to easily construct a set of invocation stack frames using Scenario 1 that execute arbitrary functions of the libc (e.g., `mmap` to map an executable memory region, `strcpy` to copy the shellcode), and an indirect control flow transfer to the injected shellcode).

If ASLR is enabled then only the location of the application image is static; all other locations like heap, all shared libraries, and the program's stack are located at randomized addresses. Exploits are limited to imported library functions and code sequences available in the application. A limitation of current ASLR implementations is that the main application object (including `data` region, `bss` region, `code` region, `got` region, and `plt` region) is mapped to a constant address. The dynamic loader resolves the dynamic locations for all imported library functions when they are called, this enables indirect calls of library functions through the `plt` slots of the application.

The imported library functions in the `plt` region can be used to resolve unimported and unreferenced functions. The location of resolved library functions are stored at static addresses in the `got` region. Gadgets can read and modify these addresses. If the binary of the library is known then the (dynamic) addresses of other library functions that are not imported can be computed. The gadget adds the offset between the imported function and the requested function to the resolved address in the `plt` slot. This enables gadgets to resolve any (unimported) library function whenever a symbol of that library is used.

5. Evaluation

The evaluation shows how the sample program in Listing 3 can be exploited under different environmental conditions. The host system uses a set of different protection features that are either enabled or disabled. The protection features are ASLR, DEP, stack protection against buffer overflows (ProPolice [14]), and if specific libc functions are already imported.

```
void foo(char *arg) {
  char text[1024];
  if (strlen(arg) >= 1024) return;
  strcpy(text, arg);
  printf(text);
}
...
foo(user_str);
...
```

Listing 3. A potential format string attack

5.1 No ASLR, no DEP, no ProPolice

If neither ASLR nor DEP are active then the format string contains a random write to, e.g., the return instruction pointer, an GOT slot, or a function pointer to redirect control flow to the injected code on the buffer on the stack. This attack conforms to the simple code injection attack in Section 2.1.

5.2 No ASLR, DEP, no ProPolice

Exploits rely on data oriented attacks if DEP is enabled. In this constellation a buffer overflow is the simplest solution to set up a ROP attack as described in Section 2.2 or a JOP attack as shown in Section 2.3.

5.3 No ASLR, DEP, ProPolice

Enabling the ProPolice extension changes the threat landscape and buffer overflows can no longer (easily) be used to exploit systems., ProPolice is enabled by default in recent versions of *gcc*[4]. On the other hand ASLR is still not enabled so a format string attack is used to redirect control flow (by overwriting the return instruction pointer of the `printf` function itself) to a gadget that adjusts the `%esp` pointer into the user-controlled string. The combination of gcc version 4.5.2 and libc-2.13 add the function `__libc_csu_init` to all compiled binaries; this function contains a gadget (`add $0x1c,%esp; pop %ebx; pop %esi; pop %edi; pop %ebp; ret` that lifts the `%esp` by 44 bytes. The format string is prepared so that it contains a set of invocation frames that enable ROP at that specific address.

Figure 2 compares the stack layouts of Listing 3 when the control flow is inside the `printf` function. ProPolice adds (i) a secure canary behind the buffer (that is checked before the return instruction pointer is dereferenced) and (ii) copies of the arguments below the buffer. The stack invocation frames in the buffer contain sets of arguments plus return instruction pointers to libc functions (e.g., a call to system would be encoded as `&system; pointer to argument string`). Pointers to, e.g., string arguments, can be directly encoded because the stack addresses are known.

5.4 ASLR, DEP, ProPolice, imports available

If ASLR is enabled then the stack and library addresses are no longer known. In this section we assume that all library functions that we want to call are imported in the main application. SOP uses the same basic technique as in Section 5.3 with two differences.

[4] ProPolice is on by default and can be disabled on request using the `-fno-stack-protector` switch.

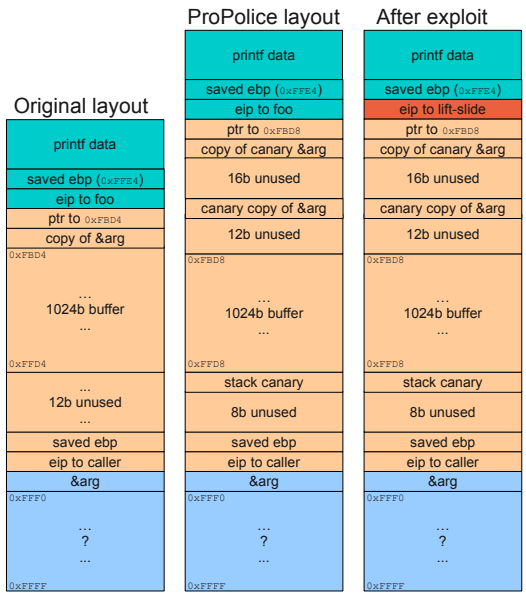

Figure 2. Stack before and after a format string based exploit that prepares stack invocation frames and works around ProPolice (blue: callee, orange: foo, green: printf).

The first difference is that the functions are not called directly but through their `plt` slots in the application (i.e., the address of `function` is replaced by the address of `function@plt`). The second difference is that addresses on the stack are no longer encoded directly but values must first be copied to well-known locations (e.g., the `bss` area of the application) using either format string writes or `strcpy`. The stack invocation frames then use the well-known locations as arguments.

5.5 ASLR, DEP, ProPolice, no imports available

If all protection mechanisms are enabled and the application does not import specific library functions then missing function addresses are resolved on the fly. Any missing imports are resolved using the approach described in Section 4.2. A single imported libc function allows an exploit to call any sequence of any libc functions with any arguments, thereby circumventing ASLR, DEP, and ProPolice.

6. Conclusion

String oriented programming (SOP) is a form of attack that relies on a format string exploit and data oriented programming. SOP is used to circumvent stack protection techniques like ProPolice using deliberate writes to memory locations whilst leaving canaries intact.

SOP then escalates to either return oriented programming or jump oriented programming to execute arbitrary code sequences without the need to inject new code, thereby circumventing data execution prevention mechanisms like ExecShield. Address space layout randomization is circumvented by copying exploit data to static regions in the memory image and resolving needed library functions through indirection attacks.

The combination of these techniques enables SOP to circumvent ASLR, DEP, ProPolice and other restrictions. An exploit uses only available data in the process image plus a format string exploit to get complete control of the application.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *CCS'05: Proc. 12th ACM conf. Computer and Communications Security* (2005), ACM, pp. 340–353.

[2] ALEPH1. Smashing the stack for fun and profit. *Phrack 7*, 49 (Nov. 1996), http://phrack.com/issues.html?issue=49&id=14.

[3] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In *ATEC '00: Proc. USENIX Annual Technical Conference* (2000).

[4] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.

[5] BHATKAR, S., BHATKAR, E., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[6] BLACKNGEL. The house of lore: Reloaded. *Phrack 14*, 67 (Nov. 2010), http://phrack.com/issues.html?issue=67&id=8.

[7] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. 6th ACM Symp. on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 30–40.

[8] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proc. 10th conf. USENIX Security Symposium* (2001).

[9] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proc. 12th conf. on USENIX Security Symposium* (2003).

[10] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proc. 7th conf. USENIX Security Symposium* (1998).

[11] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *OSDI* (2006), pp. 75–88.

[12] GADALETA, F., YOUNAN, Y., JACOBS, B., JOOSEN, W., DE NEVE, E., AND BEOSIER, N. Instruction-level countermeasures against stack-based buffer overflow attacks. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems* (2009), ACM, pp. 7–12.

[13] HAAS, P. Advanced format string attacks. https://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf, DEFCON 18 2010.

[14] HIROAKI, E., AND KUNIKAZU, Y. propolice : Improved stack-smashing attack detection. *IPSJ SIG Notes 2001*, 75 (2001-07-25), 181–188.

[15] NERGAL. The advanced return-into-lib(c) exploits. *Phrack 11*, 58 (Nov. 2007), http://phrack.com/issues.html?issue=67&id=8.

[16] OWASP. Definition of format string attacks. https://www.owasp.org/index.php/Format_string_attack.

[17] PAX-TEAM. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt.

[18] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th ACM SIGPLAN/SIGOPS Int'l conf. Virtual execution environments* (2011), ACM, pp. 157–168.

[19] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2* (2004), 20–27.

[20] PLANET, C. A eulogy for format strings. *Phrack 14*, 67 (2010), http://phrack.com/issues.html?issue=67&id=8.

[21] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th ACM conf. Computer and Communications Security* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[22] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS'04: Proc. 11th ACM conf. Computer and Communications Security* (2004), pp. 298–307.

[23] VAN DE VEN, A., AND MOLNAR, I. Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_ Execshield.pdf.