I Control Your Code
Attack Vectors through the Eyes of Software-based Fault Isolation

Mathias Payer (*mathias.payer@nebelwelt.net*)
Department of Computer Science, ETH Zurich

Abstract

Exploits are an interesting way to extend the functionality of programs. This paper presents and explains different attack vectors, namely stack-based and heap-based code injection, arc attacks on the heap as well as on the stack, format string attacks, arithmetic overflows, data attacks, and mixed ISA attacks. These attacks can be used (often in combination with other attacks) to execute arbitrary code.

From a security perspective we want to analyze how the exploit is able to hijack and redirect the control flow and what kind of malicious system calls are executed. This paper presents an approach to software-based fault isolation (SFI) that verifies every single instruction that is executed. Guards guarantee that the threat of attacks that alter the control flow, e.g., code injection, and arc attacks is removed. An additional system call authorization framework checks system calls and arguments and verifies that they conform to a strict user-defined policy. The combination of SFI and policy-based system call authorization enables an additional layer of protection for applications that runs completely in user-space.

1 Introduction

Software security is a challenging problem. Unmanaged and untyped languages are prone to code injections through stack-based or heap-based overflows. Even managed and typed languges are open to attacks using integer overflows or other data based attacks.

Attack development and attack protection is an arms race between researchers finding new ways to exploit programs and other researchers tying to come up with additional safeguards that protect from these attacks.

The main reason why exploits on current systems are so powerful is because every applications runs on a coarse-grained user-privilege level. Whenever an attacker controls an application he or she is in possession of all the privileges that the user running the application has. A tight security-model would limit the privileges on a per-application and per-user level, e.g., each application would only have access to the data of the application that is in control of that specific user.

To develop security-hardening defenses one must first understand the different attacks vectors. The goal of this paper is to present important attack vectors and analyze them in detail, including an assessment of viability and potential in the wild. The information about the different attacks enables a detailed discussion of defense mechanisms. These defenses can then be implemented to safe-guard running applications.

Safe-guards are already implemented on different levels (e.g., compiler level, operating system level, and application level). Current compilers use some static analysis to protect applications against different attacks, e.g., stack-based buffer overflows and format string attacks. Compiler-based safe-guards have the drawback that they only have a limited, static view of a dynamic application and are not able to protect against all forms of exploits. Extensions to the operating system enable policy based system call authorization, e.g., an application is only allowed to execute a subset of the available system calls. System call authorization is limited to a very coarse-grained form of security as only system calls and their arguments can be checked. The control flow inside the application remains unchecked.

A safe execution platform is able to check and validate every single instruction that is executed by the application. Every control flow transfer is verified to target a valid location and system calls are checked so that only a valid set of system calls, arguments, and locations is allowed.

This paper presents current attack vectors and potential defenses. The contributions are:

1. analysis and categorization of different attack vectors based on heap-based and stack-based code injection, arc attacks, format string attacks, and data attacks.

2. detailed protection schemes per attack vector that disable each attack and harden the security of the system.

3. trustVM as an user-space virtualization approach to safe-guard and encapsulate running applications.

2 Attack vectors

This chapter introduces the different attack vectors and discusses the necessary requirements for successful exploitation. All attack vectors have in common that they redirect control flow to new or alternate locations that would not be reached in an unaltered run. Control flow can be altered through buffer overflows, format string attacks, or data attacks.

All attack vectors exploit the fact that the programmer or the runtime system was unable to check the bounds of a buffer or to detect a type overflow (e.g., integer overflow) to overwrite either data structures or code.

2.1 Code injection

Code injection attacks place new machine code in the running application and redirect the control flow to the newly placed code. This code then executes the malicious payload.

2.1.1 Stack-based code injection

This attack exploits missing or incomplete bound checks of a local array on the stack. The array is filled with user-controlled code. Because the stack grows downwards it is possible to overwrite the variables that are higher up on the stack, including the saved base pointer and the saved return instruction pointer. The attack adjusts the return instruction pointer so that it points to the code that was placed in the buffer. The function therefore does not return to the caller but to the code on the stack.

Listing 1 shows a simple code sequence that is prone to a stack-based overflow where code injection is possible. Figure 1 shows the stack layout before and after an overflow. The length of the user supplied input overflows the length of the variable on the stack and overwrites the base pointer and the return instruction pointer. The new return instruction pointer then points to the beginning of the temporary array on the stack. The exploit code is executed when the function returns.

```
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

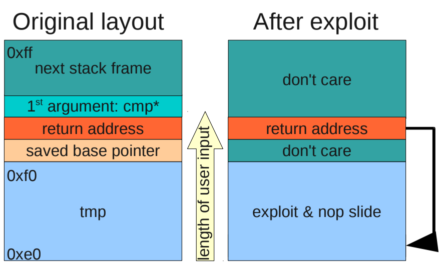

Figure 1: Stack before and after a stack-based code injection exploit.

Constraints for this attack are that code on the stack must be executable, a bound check for a buffer on the stack must be missing or faulty, and the runtime system may not verify the return instruction pointer.

2.1.2 Heap-based code injection

This form of attack places the malicious code in an array on the heap and redirects control flow to the injected code. The control flow redirection is achieved through overwriting the return instruction pointer, overwriting a function pointer (or vtable entry for C++), adjusting GLIBC destructors, or fiddling with the memory allocator datastructures.

Listing 2 shows a simple code sequence with a vulnerable `struct` and a code sequence that enables a heap-based code injection. Figure 2 shows the vulnerable struct before and after an exploit. The buffer in the vulnerable struct is filled with a user supplied nop-slide[1] and exploit code. The function pointer is overwritten and points somewhere into the nop-slide of the struct's buffer. The exploit code is executed when the function pointer is called.

```
typedef struct vuln_struct {
  char buf[MAX_LEN];
  int (*cmp)(char*);
};
int is_foobar_heap(vuln_struct *s, char *str) {
  // assert(strlen(cmp) < MAX_LEN)
  strcpy(s->buf, str); // no bound check
  return s->cmp(s->buf, "foobar");
}
...
vuln_struct *st = \
  (vuln_struct*)malloc(sizeof(vuln_struct));
// user_str is > MAX_LEN
if (is_foobar_heap(st, user_str))
...
```

Listing 2: A potential heap-based overflow

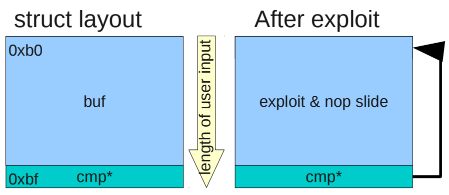

Figure 2: Vulnerable struct before and after a heap-based code injection exploit.

The location where the code is placed must be executable and a bound check for a buffer must be faulty or missing. Additionally the control flow must be redirected to the code placed on the heap.

[1] A sequence of nop instructions is used if only a vague location is known. The exploit transfers control somewhere into the nop-slide.

2.2 Arc attacks

Arc attacks or return based programming uses already existing code sequences to execute malicious payload. A stack-based overflow is used to prepare the stack so that tails of library functions are executed one after another. These code sequences are aligned and prepared in such a way that arbitrary code execution is possible.

A stack-based buffer overflow as in Listing 1 can be used to prepare the stack for an arc attack. Figure 3 shows a simple arc attack that uses a buffer overflow on the stack. The buffer and the saved base pointer are overwritten with garbage data, the return instruction pointer is redirected to a function in the glibc (`system()` in this case). The function pointer is followed by the saved `ebp` that is used when the function returns and the arguments to that function.

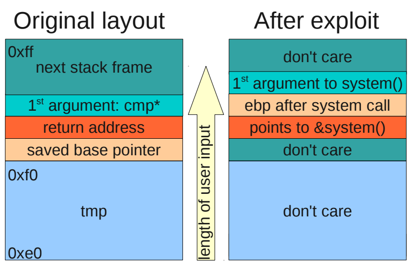

Figure 3: Stack before and after a stack-based arc injection.

This attack relies on a stack-based overflow and that the return instruction pointer is not verified before it is dereferenced.

2.3 Format string attacks

A format string attack exploits the argument parsing possibilities of the `printf` family. If an unchecked user controllable string is passed to a `printf` function then the attack can read and write arbitrary memory addresses and pop an arbitrary amount of data from the stack.

Listing 3 shows code with a potential format string exploit. Using a combination of `%x` to increase the amount of printed characters and `%n` that writes the number of already printed characters to a given location it is possible to read and write arbitrary memory locations. Assume that in Listing 3 the return instruction pointer is at `0xffffd37c` and the text array is 11 words higher up on the stack relative to the printf call. An input string of `(0xffffd37c)(0xffffd37e)%12$2043x.%12$hn%11$32102x%11$hn` will overwrite the return instruction pointer with `0x0804856a` by writing two half words namely $0x0804 - 9 = 2043$, and $0x856a - 2043 = 32102$.

Depending on the environment a format string exploit overwrites the return instruction pointer, the global offset table (GOT), or the list of destructors. All of these variations alter the control flow at one point in time.

```
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

The requirement is that the user string must contain escape sequences like `%n` or `%s` that are then parsed and expanded by `printf`.

2.4 Arithmetic overflow

Arithmetic data types always have specific bounds. An 1 byte data type can only store 256 different values. If an operation (e.g., addition) exceeds these bounds then the variable wraps around and continues on the other end (e.g., $127 + 1 = -128$ for an 1 byte, signed data type, or $255 + 1 = 0$ for an unsigned 1 byte data type). These overflows can be used to bypass bound checks (e.g., before memory is allocated).

Listing 4 shows a potential arithmetic overflow. If len has the value 0x40000000 then $len > 0$ holds but the result of the multiplication in the malloc call is 0. The following memcpy will overwrite data structures on the heap, resulting in a heap-based overflow.

```
void foo(int len, char *pack) {
  char *response;
  if (len > 0) {
    response = malloc(len*sizeof(char*));
    memcpy(response, pack, len);
  }
}
...
foo(user_len, user_packet);
...
```

Listing 4: A potential arithmetic integer overflow

Requirements for an arithmetic overflow are lax or implicit type conversions, sign errors, rounding errors, type overflows due to arithmetic operations, and pointer arithmetic.

2.5 Data attacks

A data attack exploits a missing or faulty bound check to write data to an user-controlled address. This random write is used to redirect control flow to injected code or to set up a secondary attack.

```
void foo(int pos, int value, int *data) {
  data[pos] = value;
}
...
foo(user_pos, user_value, data);
...
```

Listing 5: A potential data attack

Listing 5 is prone to a potential data attack. The attacker controls the position and the value that is written. A simple calculation relative to the position of the data array enables a random write to (almost) any memory location. Most real programs have some checking that limit the location and value that can be written.

Requirements for a data attack are unfiltered or only partially filtered user input and missing or faulty bound checks.

2.6 Mixing x86_64 and i386 code

Modern operating systems support x86_64 and i386 code in parallel. An application can even use both modes interchangeably. Mixing x86_64 and i386 code is used to trick static verifiers or to escalate privileges. Most static verifiers and system call authorization frameworks are limited to either x86_64 or i386 code. An application that uses both instruction sets to execute system calls or specific control transfers is able to escape the control of the guards [10].

3 Protection

User-space software-based fault isolation (SFI) with additional guards can protect a running application from all the attacks described in Section 2. SFI separates a running application into two user-space protection domains, the virtualization layer and the application layer. The virtualization layer controls every single control flow transfer of the application and checks all instructions before they are executed. The virtualization system caches executed code in a code cache to reduce the overhead of virtualization. The virtualization system adds additional guards to the translated code and verifies static control flow transfers during the translation process. Indirect control flow transfers, where the target is not known at translation time, are wrapped into an additional runtime check.

3.1 Software-based fault isolation

Any dynamic binary instrumentation tool like HDTrans [22, 21], DynamoRIO [5], Pin [14], Valgrind [16], or fastBT [18, 17] can be used to implement SFI. The binary translation framework must support the following features:

1. return addresses on the stack and indirect jump targets must point to the original location. This feature separates the binary translator and the running application and implements a protection domain in user-space.

2. the full x86_64 and i386 ISA must be supported, otherwise an exploit can target the missing instructions and escape the binary translator.

3. the translation and runtime overhead should be low. The chances of wide adoption is low if the overhead for SFI is high.

4. a small trusted computing base enables the verification of the binary translator's code.

We use fastBT, a low-overhead, small, dynamic binary translation system that supports the complete i386 and x86_64 ISA, to implement additional guards that guarantee SFI. fastBT is the only binary instrumentation framework that supports all features described above. The SFI framework wraps application code into an additional layer of protection. The additional guards verify all code locations. Only verified and translated code is executed.

The additional guards execute safety checks during the translation process of dynamic code for static targets and add runtime checks where a static target verification is not enough (e.g., for dynamic indirect control flow transfers). Figure 4 gives an overview of the translation process and shows the different stages where guards are inserted.

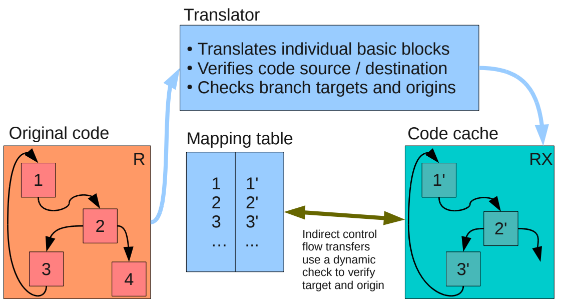
Translator
• Translates individual basic blocks
• Verifies code source / destination
• Checks branch targets and origins
Original code
R
1
2
3
4
Mapping table
1 1'
2 2'
3 3'
… …
Indirect control flow transfers use a dynamic check to verify target and origin
Code cache
RX
1'
2'
3'

Figure 4: Overview of the translation process. The guards are executed during the translation phase and whenever indirect control flow transfers are dispatched.

All system calls are wrapped into an additional protection framework. This system call authorization framework checks each system call based on system call number, location of the system call, and given parameters. These authorization parameters are specified using a simple policy. The policy based authorization system is extended by system call guards that execute specific handler functions to examine the parameters of a system call in detail (e.g., `open` system calls are checked for path arguments, `mprotect` system calls are checked for executable and writeable flags, and `mmap` system calls are checked for already mapped regions and overlaps).

Figure 5 shows the additional layer of protection that is wrapped around the running application. The sandbox protects the user and the system from bugs of the application.

3.2 Code injection

The SFI framework records all code locations of the original application and marks these locations as read only. All executed code is translated first and the SFI framework checks the location of translated code. If the

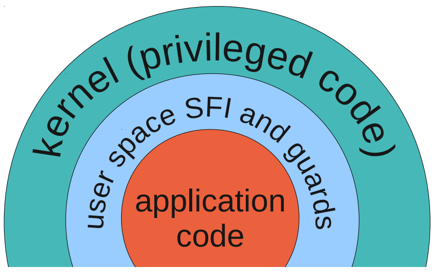

Figure 5: Sandbox overview of an application running under the control of user-space fault isolation.

translated code is not part of the original code regions then the program is aborted.

Because all original code locations are marked read only new code must be injected to data regions. These data regions lack the executable bit and the translator aborts when a control flow transfer to a data region is detected.

This guard protects from all forms of code injection. Only code that is available in the application is allowed to execute.

3.3 Arc attacks

Arc based attacks redirect control flow to already existing code chunks in the application. Stack-based arc attacks [20] use artificial stack frames to execute arbitrary code using function tails. Heap-based arc attacks redirect control flow to unintended code (by, e.g., overwriting a function pointer, or a vtable).

The SFI framework detects the redirection of the control flow into a function of a different module. Inter-module control flow transfers must target exported functions. If an inter-module control flow transfer targets a non-exported symbol then the application is terminated. This guard stops all arc attacks that redirect control flow to code sequences inside functions.

3.4 Data attacks and arithmetic overflows

Data attacks and arithmetic overflows result in a (more or less random) write to memory. Data driven attacks that redirect control flow to injected code are caught by the code verification guards from Section 3.2. Attacks that only change data are not detected by the SFI framework but are detected whenever a system call is executed that is not part of the policy or if the arguments of the system call exceed the policy's privileges.

4 Related work

Security can be enforced on many different levels and through different approaches. This section describes orthogonal or similar approaches to SFI as well as SFI itself. Protection and verification is either done ahead of time or dynamically at runtime. Ahead of time protection reduces the potential overhead but either restricts the ISA or uses complicated static analysis. Dynamic protection checks code during the execution using dynamic runtime safety checks. The granularity of protection differs widely between approaches. A virtual machine operates at a very coarse-grained level of protection with additional overhead to provide a complete system image per application. Protection mechanisms on the system call level are less coarse-grained but only user-space protection mechanisms enable the complete control over all executed instructions, including control flow transfers.

4.1 Compiler extensions

Compiler extensions are able to protect the code when it is translated to machine code. The compiler adds additional runtime checks for, e.g., format strings, buffer allocations, and return instructions to guard the running application. Static verification reduces the amount of necessary runtime checks. Examples for such systems are, e.g., StackGuard [9], FormatGuard [6], Propolice [13], PointGuard [7], and libverify [3].

Each of these systems removes (or reduces) one attack vector. The drawback of these approaches is that they do not consider the complete picture and only concentrate on one single attack form, leaving other attack vectors open.

4.2 System call authorization

System call authorization offers a coarse-grained form of protection on the system call level. Individual system calls of an application are enforced according to a static or dynamic policy. Examples of such systems are AppArmor [4], Linux Security Modules [23], Systrace [19], MAPbox [1], SubDomain [8], Switchblade [11], and Consh [2].

The drawback of these systems is that code exploits inside of an application are still possible. The application is only bound by the policy and can execute any system calls and arguments that are allowed. An exploit is able to circumvent these sandboxes if the system call policy is not tight enough. A secure runtime system can use system call authorization as a second line of defense, but should not rely on system call authorization to catch all possible exploits.

4.3 Static verification

Static verification processes the application's code and guarantees statically that no exploits are possible. These static verifiers limit the ISA in several ways, e.g., they pad instructions so that they do not cross 16 byte boundaries, they align indirect control flow transfers, and limit the interaction possibilities with the operating system. Such approaches are implemented in PittSFIeld [15] and Google's Native Client (NaCl) [24].

4.4 Software-based fault isolation

Software-based fault isolation uses dynamic binary translation to add dynamic guards to the executed code.

These guards guarantee the safe execution of applications and terminate the application in case of an error. Vx32 [12] is a system that implements SFI.

A drawback of these systems is that they only cover code based exploits and ignore data based exploits. Only a combination of SFI and system call authorization is able to cover all kinds of exploits.

5 Conclusion

Many different forms of attacks exist, like stack-based and heap-based code injection attacks, arc attacks on the stack and on the heap, format string attacks, arithmetic overflows, data attacks, and mixed ISA attacks. These attacks are often combined and used to exploit a running application. Our software-based fault isolation approach enables an additional layer of virtualization and encapsulates the running application. Every machine code instruction is verified and all control flow transfers must target valid code. Additionally all system calls, including the individual arguments, must conform to a strict system call policy.

The combination of software-based fault isolation and system call authorization is an important tool to increase the level of security. Dynamic instrumentation is an interesting tool to understand exploits and to protect running applications against known and unknown forms of attacks.

The source code of the fastBT virtualization framework can be downloaded at *http://nebelwelt.net/fastBT*.

References

[1] ACHARYA, A., AND RAJE, M. MAPbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium* (2000).

[2] ALEXANDROV, A., KMIEC, P., AND SCHAUSER, K. Consh: Confined execution environment for internet computations, 1999.

[3] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent runtime defense against stack smashing attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference* (2000).

[4] BAUER, M. Paranoid penguin: an introduction to novell apparmor. *Linux J. 2006*, 148 (2006), 13.

[5] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (Washington, DC, USA, 2003), pp. 265–275.

[6] 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: Proceedings of the 10th conference on USENIX Security Symposium* (2001).

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

[8] COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. Subdomain: Parsimonious server security. In *LISA '00: Proceedings of the 14th USENIX conference on System administration* (2000).

[9] 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: Proceedings of the 7th conference on USENIX Security Symposium* (1998).

[10] EVANS, C. Linux kernel "seccomp" facility minor vulnerability. CESA-2009-004.

[11] FETZER, C., AND SUESSKRAUT, M. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM, pp. 273–286.

[12] FORD, B., AND COX, R. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 293–306.

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

[14] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05* (New York, NY, USA, 2005), pp. 190–200.

[15] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (Vancouver, BC, Canada, August 2–4, 2006), pp. 209–224.

[16] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (New York, NY, USA, 2007), pp. 89–100.

[17] PAYER, M., AND GROSS, T. Requirements for fast binary translation. In *2nd Workshop on Architectural and Microarchitectural Support for Binary Translation* (2009).

[18] PAYER, M., AND GROSS, T. R. Generating low-overhead dynamic binary translators. In *SYSTOR'10* (2010).

[19] PROVOS, N. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), USENIX Association, pp. 18–18.

[20] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[21] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[22] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE '06* (New York, NY, USA, 2006), pp. 175–185.

[23] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (2002).

[24] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. *IEEE Symposium on Security and Privacy* (2009), 79–93.