Library-Level Policy Enforcement

Marinos Tsantekidis
Vassilis Prevelakis
TU Braunschweig
Institute of Computer and Network Engineering
Email: {tsantekidis, prevelakis}@ida.ing.tu-bs.de

***Abstract*—We propose a system that allows policy to be implemented at the library call level. Under our scheme, calls to libraries are monitored and their arguments examined to ensure that they comply with the security policy associated with the running program. Our system automatically creates wrappers for libraries so that calls to external functions in the library are vectored to a policy enforcement engine. In this paper, we describe our system, which screens calls to protected functions, while allowing the implementation of a high level form of control flow integrity based on library calls. It is a transparent approach that can protect applications in many different domains and real-life environments.**

Keywords*—*policies; library calls; argument examination; wrapper functions

I. INTRODUCTION

Access control, in a narrow sense, is the ability of a system to grant or reject access to a protected resource. This way, in the context of software security, the system can keep track of who has access to what code, who can call what function in a library and under which conditions this is possible. These restrictions are imposed by a set of mandatory controls that are enforced by the system in the form of policies. Policies may represent the structure of an organization or the sensitivity of a resource and the clearance of a user trying to access it. A mechanism maps a user's access request to a collection of rules that need to be implemented in order for the system to function in a secure manner.

An access control system can be implemented in many places and at different levels in an infrastructure (e.g., operating system, database management system, etc.) and must be configured in a way that provides the assurance that no permissions will be leaked to an unintended actor, which may give her the ability to circumvent any defenses in place.

In this paper, we propose a novel mechanism that aims to allow access control policies for library calls to be enforced at the user-code level in order to restrict access to functions held in a protected library, in addition to identifying the complete execution path regarding the functions in question. At runtime, the policy system may be used for policy enforcement. It can coexist with existing defense techniques, boosting the security of the protected system.

The remainder of this paper is organized in the following manner: Section 2 describes related work done to address relevant issues. In Section 3, we present the architecture of our framework. In Section 4, we describe the implementation details, along with possible applications. In Section 5, we present a simple use-case scenario. Section 6 concludes this paper.

II. RELATED WORK

This work revisits our earlier work on Access Controls For Libraries and Modules (SecModule) [1]. This framework forces the user-level code to perform library calls only via a library policy enforcement engine providing mandatory policy checks over not just system calls, as in the case of Systrace [2], but calls to user level libraries as well. This results in a system which can be used to systematically formulate and formalize rights management for software. The access rights in question would be whether a process (which may be malicious) is allowed to execute some function held securely in a library module. Initially, the mechanism retrofitted functions in order to be included in a secure "enclosure" (SecModule). The kernel has a list of all the SecModules and when a process asks for access to a secured function, the kernel verifies that the requested SecModule is registered and that the process is valid with respect to its policy. Then, it allows that and only that process to use only the specific function.

This means that access to a specific function or procedure is controlled by the kernel. While this is particularly suited to SecModule-enabled applications, the overhead of two context switches per function invocation (once to transfer control to the kernel and – when it reaches a decision – once more to transfer control back to the caller) makes the technique quite expensive for more general use.

One of the issues identified by the authors of the SecModule paper was the difficulty in encapsulating library modules. This manual process was error prone and extremely labor intensive, since most of the applications compiled within the framework required patching. Another issue was the inability to evaluate call arguments. Although they were contained in a known structure pointed to by a stack pointer, their examination required lots of casting in the C++ functions, which in turn needed additional information for these functions held in the module.

Relevant to our work is the Systrace [2] system which supports fine-grained policy generation. It guards the calls to the operating system at the lowest level, enforcing policies that restrict the actions an attacker can take to compromise a system. In the process, although, it makes higher level actions indistinguishable. As an example, we can look at `libancillary` [3], a tiny library that provides an interface to operations that can be done on Unix domain sockets.

Programs that use this library can send/receive one or more file descriptors to/from a socket, actions particularly useful when the primary process lacks the rights required to open a file or a device. In this case, another – privileged – process opens the resource and sends a corresponding file descriptor to the requesting process for further processing. To control this exchange and prevent arbitrary usage of this library, the system calls (`open(2)`, `send(2)`, `recv(2)`) would need to be examined and policies enforced under Systrace. However, these calls result in a number of lower level calls to the operating system all of which Systrace would need to check. Since only the high-level calls are of importance in this case, the examination of underlying calls would be not only unnecessary, but unwanted too. The fine-grain control offered by the framework while checking calls required by system or user level libraries when implementing complex operations, is overly verbose. Additionally, it may leave a library in an inconsistent state if the sequence of these calls is interrupted in the middle of execution by a misconfiguration [1]. Furthermore, for applications that use high-level abstractions away from low-level system calls, there may be difficulties generating precise policies. Later research [4] showed that concurrency vulnerabilities were discovered that gave an attacker the ability to construct a race between the engine and a malicious processes to bypass protections. More specifically, in a multiprocessor environment the arguments of a system call were stored by a process in shared memory. After Systrace performed the check and permitted the call, another malicious process had a time window to replace the cleared arguments in shared memory, effectively negating the presence of Systrace and evading its restrictions. In a uniprocessor environment, this could be achieved by forcing a page fault or in-kernel blocking so the kernel would yield to the attacking user process.

Multics [5] operating system uses multiple rings of protection [6] – [7] that isolate the most-privileged code from other processes, forming a hierarchical layering. Each process is associated with multiple rings – domains – so it is necessary to change the domain of execution of a process. This way the process can access specific domains only when particular programs are executed. To prevent arbitrary usage, specific "gates" between rings are provided to allow passing from an outer (less-privileged) to an inner (more-privileged) ring, restricting access to resources of one layer from programs of another layer. The change of domain occurs only after the control is transferred to a gate of another domain. Switching to a lower ring requires more access rights as opposed to a higher ring where reduced rights suffice. Downward switching requires a control transfer to a gate of an inner ring, if the transfer is to be allowed, whereas an upward domain switch is an unrestricted transfer that can be performed by any process. Nevertheless, the need-to-know principle cannot be enforced, because if a resource needs to be accessible by a ring `a` but not from another `b`, then `a` needs to be lower than `b`. But, in this case every resource in `b` is accessible in `a`.

Similar to our mechanism, `ltrace` [8] [9] is a utility that runs a specified command until it exits. It intercepts the calls made to shared libraries by an application and displays the parameters used and the values returned by the calls. Moreover, it can trace system calls executed by the application. However, because it uses the dynamic library hooking mechanism, it cannot trace statically linked executables/libraries, as well as libraries that are loaded automatically using `dlopen(3)`. This mechanism gives the programmer the ability to inject symbols in the dynamic library, but these symbols need to be unresolved in the main executable or be exported in its dynamic symbol table. When the linker tries to resolve them, it will find the injected symbols and not the original ones. A statically linked application has neither unresolved symbols nor a dynamic symbol table. Additionally, `ltrace` can only display the parameters used and values returned by the calls. It offers no ability to manipulate them.

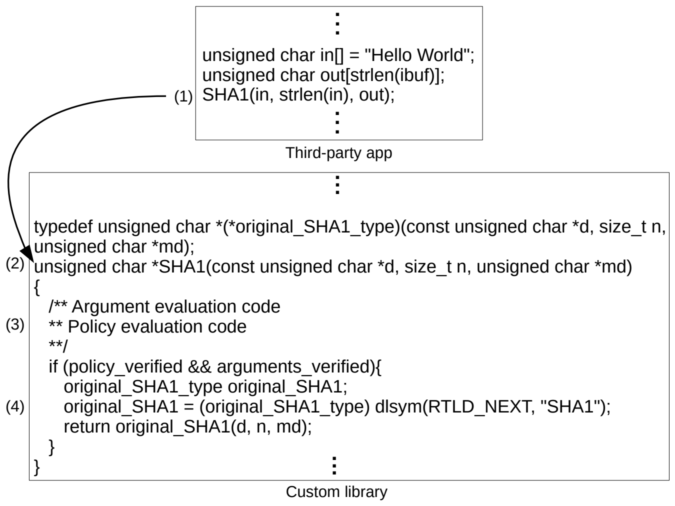
Abadi et al. proposed CFI [10] which enforces the execution of a program to adhere to a control flow graph (CFG), which is statically computed at compile time. If the flow of execution does not follow the predetermined CFG, an attack is detected. This approach, however, suffers from two main disadvantages. First, the implementation is coarse-grained. Computing a complete and accurate CFG is difficult since there are many indirect control flow transfers (jumps, returns, etc.) or libraries dynamically linked at run-time. Furthermore, the interception and checking of all the control transfers incur substantial performance overhead.

In our work, we also implement access control similar to the work presented above. Under our scheme, each call to an external function of a library is intercepted and checked to ensure that it complies with the security policy associated with the running program. Every time a call to such a function is made, its arguments are examined and it is vetted by policy evaluation code to determine whether the control flow transfer is warranted. This allows high-level policy checks to be carried out in a similar fashion to the Systrace engine [2] – which, however operates at the system call level – and SecModule [1] that introduced a form of authentication when calling functions from a library.

III. Design

Our system aims to automate the process of encapsulating library modules and allow entire libraries to be instrumented, checking the arguments of the calls to functions within a library along the way, before reaching a policy decision. The flow of execution inside the protected library can also be detailed, revealing the sequence of calls to its functions. Figure 1 depicts a high level overview of the steps taken when an untrusted app calls a protected function.

In step (1), the application calls a function secured in our custom library (in this case SHA1). In step (2), instead of the intended function, the secure wrapper version of it is executed. Instrumented inside the wrapper, there is argument and policy evaluation code, which is first run before any other steps are taken (step 3). If the evaluation is successful, the originally



```
⋮
unsigned char in[] = "Hello World";
unsigned char out[strlen(ibuf)];
SHA1(in, strlen(in), out);
⋮
```

```
⋮
typedef unsigned char *(*original_SHA1_type)(const unsigned char *d, size_t n,
unsigned char *md);
unsigned char *SHA1(const unsigned char *d, size_t n, unsigned char *md)
{
   /** Argument evaluation code
   ** Policy evaluation code
   **/
   if (policy_verified && arguments_verified){
      original_SHA1_type original_SHA1;
      original_SHA1 = (original_SHA1_type) dlsym(RTLD_NEXT, "SHA1");
      return original_SHA1(d, n, md);
   }
}                                          ⋮
```

Figure 1. Overview of the call sequence

intended function is called (step 4) and the execution continues normally.

Due to the fact that we interject our evaluation code between the original call and the intended function, our approach is transparent. It requires no code modifications on the library's code, which makes it suitable for legacy applications. Also, it can be used on binary programs, since there is no need to have access to or recompile the source code of the application.

The product of the customization of a library – which is a shared custom library – can be easily adopted by security experts and used in real-life environments, since it only needs to be preloaded before running an application.

No context switch is necessary, using our custom library, since the kernel is not invoked in anyway whatsoever. Contrary to SecModule, our technique is inexpensive that way. Furthermore, the encapsulation of the library functions is straightforward using just a python script to automate the procedure, requiring only minimal manual intervention. Past experience of the writers and simplicity in producing the code, as well as major support from the community, lead to the decision of using Python as the means to create the shared library.

Under our scheme, the parameters of the intercepted calls can not only be observed, but also manipulated in order to be sanitized if necessary. Unlike `ltrace`, our mechanism relies on `dlsym(3)` and `dlopen(3)` to find the address of a symbol in memory, but because it also relies on dynamic library hooking, it is unsuitable for tracing statically linked applications.

Based on our current approach, the size of the code is increased because extra code needs to be added for every function. Before making the intended call, an extra wrapper is executed in order to decide whether to redirect the flow to the initial call or not.

Additionally, our framework depends on the programming language used to develop the protecting application, since – currently – it can only protect applications written in C/C++.

Furthermore, if an attacker knows of the presence of the protection mechanism, he might be able to bypass the policy evaluation step and call the intended function directly. Nevertheless, we believe that randomization techniques, such as ASLR [11], will make direct calls to libraries untenable.

IV. Implementation

Our technique aims to monitor calls to external functions inside a protected library. We investigated two ways of doing this: (a) individual wrappers or (b) one overall wrapper

- In the first approach, we install separate wrapper functions. Each function in the library that has an interface to the outside world is enclosed in a wrapper. When the wrapper is called, first it executes policy evaluation code to determine if the caller is permitted to call the function and then redirects the flow to the originally intended function or not.
- In the second case, the wrapper stands at the entry point of the library. A policy enforcement engine inside the wrapper monitors the incoming requests and when a call is made to a function, it determines whether that call is warranted (i.e., in accordance to the system policies). It then diverts the flow of execution to the called function.

In both approaches, the policy evaluation code examines the arguments of the call to ensure that they comply with the security policy associated with the running program.

In this first version of our prototype, we decided to follow the first path, due to the simplicity of the implementation. As an example, we created a wrapper for the OpenSSL library. The header files of the library can be included in any C/C++ program by the developers and contain all the functions that they can call. First, we extracted from the header files all the relative functions and their signatures. The extraction was done using a custom Python script that identifies each function that is within the scope of our work and analyzes its arguments. This way we are able to manipulate each of the arguments in any way necessary. Before calling the originally intended function we added code that verifies that the module, indeed, captured the call and that we operate from within the custom library. After implementing the security features (i.e., argument examination, policy enforcement, etc.) and if the continuation of the execution is permitted, the flow progresses to the original path. The result is a C file that is compiled to a shared library which is preloaded when running a program.

Automatic generation of policy (learning phase) will also be supported in future versions, while at run-time the policy system will be used for policy enforcement and/or for ensuring that the program behaves in a similar manner as in the learning phase. During this phase, as many as possible execution paths will need to be discovered, that correspond to actions taken from a benign application, aiming to implement a CFI [10] scheme that uses library calls to extract execution paths, instead of intercepting or instrumenting or emulating the control flow instructions. This will form a basis on top of which more complete policies will be built.

Applications

Wrapped functions can be accessed in a controlled manner via mandatory policy checks prior to the execution of the original flow. When an attacker tries to manipulate the protected library, the malicious efforts will be thwarted since they do not conform with the policies enforced. Nevertheless, our code can be bypassed if the attacker knows of its existence and calls the original library function directly. However, within the SHARCS project [12], we are working on hardware primitives that will force the user code to go through our wrapper.

Digital Rights Management (DRM) is another domain that our framework can be used for. In this context, it can provide access control in order to restrict usage of a piece of software the owner of which retains the right to distribute on his own conditions (e.g., after getting some form of payment or even just recognition for his efforts) or prevent the theft of it.

In the case of a library that requires heavy resources from the host system, the administrator may wish to control access to the rights to invoke the code in such a way that the system does not hang by over-use or is not affected by a DDoS attack. Access restrictions can be imposed according to certain criteria or security policies enforced by an organization.

The misuse of a critical component in a secure infrastructure can result in unforeseen consequences for the system. Our framework can make sure that only authorized personnel can have access to the secure part. Even in the case of deliberate actions that lead to an attack that jeopardizes the system, our framework can be used as a logging mechanism. The inner workings of a protected library will be traced, which will follow the flow of execution of functions held within the library. Forensic actions (after the fact) can, then, be taken to analyze in a more detailed view the events that led to the compromise and identify the culprits responsible.

V. USE-CASE STUDY

In this section, we present a scenario where a vulnerability of an application is exploited to affect the availability of the system. In our use-case, we use a vulnerable version of OpenSSL library, where a buffer overflow is triggered under specific circumstances to launch a DoS attack, in order to crash the application. By using our instrumented library to observe calls to the OpenSSL functions, we can better understand the behavior of the attack and characterize the vulnerability.

A. ChaCha20-Poly1305 heap buffer overflow

CVE-2016-7054 [13] [14] is a recent heap-based buffer overflow vulnerability related to TLS connections using *-CHACHA20-POLY1305 cipher suites. It was discovered on September 2016 and characterized as highly severe. Servers implementing versions 1.1.0a or 1.1.0b of OpenSSL, can crash when using the ChaCha20-Poly1305 cipher suite to decrypt large payloads of application data, making them vulnerable to DoS attacks. It is triggered by an error during the verification of the MAC. If it fails, the buffer on which the decrypted ciphertext is stored, is cleared by zeroing out its content via the `memset` function. However, the pointer to the buffer that is passed to the function points to the end of the buffer instead of the beginning. If the payload to be cleared is large enough, the contents of the heap will be erased, resulting in a crash when OpenSSL frees the buffer.

B. Custom library implementation

Although the vulnerability described in the previous section was addressed in versions later than 1.1.0b, we can use our prototype to examine the chain of events inside the OpenSSL library that result in a crash when the vulnerability is exploited.

When we first start an OpenSSL server (e.g., `LD_PRELOAD=/home/user/Desktop/custom_lib.so ./bin/openssl s_server -cipher 'DHE-RSA-CHACHA20-POLY1305' -key cert.key -cert cert.crt -accept 4433 -www -tls1_2 -msg`), an initialization phase takes place, where we can see that memory is allocated for the `s_server` app. Excerpt from our framework:

....
Intercepted call to function CRYPTO_strdup
String parameter: apps/s_server.c".
....

Then, the private key and certificate files are read. Excerpt:

....
Intercepted call to function BIO_new_file
String parameter 1: cert.key
String parameter 2: r
....
Intercepted call to function BIO_new_file
String parameter 1: cert.crt
String parameter 2: r
....

After that, a pointer to every cipher supported by TLS v1.2 is pushed on the cipher stack, if it is not already there. Excerpt:

....
Intercepted call to function EVP_add_cipher
Intercepted call to function EVP_aes_256_ccm
Intercepted call to function EVP_add_cipher
Intercepted call to function EVP_aes_128_cbc_hmac_sha1
....

Continuing in a similar manner, a pointer to every message digest supported by TLS v1.2 is pushed on the digest stack, if it is not already there. In addition, aliases are mapped to ciphers/digests. Excerpt:

....
Intercepted call to function EVP_md5
Intercepted call to function EVP_add_digest
Intercepted call to function OBJ_NAME_add
String parameter 1: ssl3-md5
String parameter 2: MD5
Intercepted call to function EVP_add_digest
Intercepted call to function EVP_sha1
....
Intercepted call to function OBJ_nid2sn
Intercepted call to function EVP_get_cipherbyname
String parameter: DES-EDE3-CBC

....

Then, memory is allocated based on the compiled-in ciphers and aliases. Excerpt:

....
Intercepted call to function CRYPTO_malloc
String parameter: ssl/ssl_ciph.c
Intercepted call to function FIPS_mode
....

At the end of this initialization process, an "ACCEPT" message is displayed, notifying the user that the server is up and running and awaits incoming connections. Excerpt:

....
Intercepted call to function BIO_printf
String parameter: ACCEPT
....

To automate our efforts we used an open-source, python, TLS test suite and fuzzer named tlsfuzzer [15] which includes a script to exploit CVE-2016-7054.

When the script is executed, we see a number of calls to `BIO_printf` function which display the messages exchanged between client and server (ClientHello, ServerHello, ServerKeyExchange, etc.). Then, at some point during execution, we see a call to `ERR_put_error` which signals that an error occurred and adds the error code to the thread's error queue. Excerpt:

....
Intercepted call to function ERR_put_error
String parameter 1: ssl/record/ssl3_record.c
....

Continuing, the program gets the error's code from the queue via `ERR_peek_error`. Then `ERR_print_errors` is called to print the error string. At this point, memory is freed via calls to functions like `CRYPTO_free`, `BIO_free_all`, `CRYPTO_free_ex_data`, `OPENSSL_cleanse`, `EVP_CIPHER_CTX_free` etc. Under normal circumstances, the server would reset the connection awaiting new incoming messages, but due to the CVE-2016-7054 bug the heap is nullified and the sever crashes, potentially indicating a DoS attack.

During the exploitation of this vulnerability, our library shows all the system calls made from the phase of the initialization of the server, to the handshake between it and the client, to the crash after the attack. This can provide a forensic trail to identify the functions executed in the OpenSSL session, in order to pinpoint where the vulnerability is triggered – in this case, when the memory is freed.

VI. Conclusion

In this paper, we presented an access control scheme that produces custom libraries and examines calls to functions within them along with their arguments, to ascertain if they adhere to specific security policies. Our framework improves important aspects of SecModule in which it can be incorporated, simplifying and automating the generation of libraries and providing a seamless way of evaluating the arguments of each call.

Our approach is transparent and can be used on binary/legacy applications and existing environments, as well as serve as a complimentary measure of defense alongside already implemented mechanisms.

Acknowledgement

This work is supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571. Additionally, it is supported by the DFG Research Unit, Controlling Concurrent Change (CCC) project, funding number FOR 1800.

References

[1] J. W. Kim and V. Prevelakis, "Base line performance measurements of access controls for libraries and modules," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 356–356. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898699.1898911 [accessed: 2017-07-26]

[2] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251371 [accessed: 2017-07-26]

[3] N. George, "Ancillary library." [Online]. Available: http://www.normalesup.org/ george/comp/libancillary/ [accessed: 2017-07-26]

[4] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *Proceedings of the First USENIX Workshop on Offensive Technologies*, ser. WOOT '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2:1–2:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1323276.1323278 [accessed: 2017-07-26]

[5] "Multics." [Online]. Available: http://www.cse.psu.edu/˜trj1/cse443-s12/docs/ch3.pdf [accessed: 2017-07-26]

[6] "Multics rings," 1996. [Online]. Available: http://www.cs.unc.edu/ dewan/242/f96/notes/prot/node11.html [accessed: 2017-07-26]

[7] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," 1972. [Online]. Available: ftp://ftp.stratus.com/vos/multics/tvv/protection.html [accessed: 2017-07-26]

[8] J. Cespedes, "ltrace." [Online]. Available: https://linux.die.net/man/1/ltrace [accessed: 2017-07-26]

[9] "ltrace." [Online]. Available: http://www.ltrace.org/ [accessed: 2017-07-26]

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165 [accessed: 2017-07-26]

[11] T. PaX, "Address space layout randomization," 2001. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt [accessed: 2017-07-26]

[12] SHARCS, "Secure hardware-software architectures for robust computing systems," 2015. [Online]. Available: http://www.sharcs-project.eu/ [accessed: 2017-07-26]

[13] CVE_2016_7054, "Chacha20/poly1305 heap-buffer-overflow," 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054 [accessed: 2017-07-26]

[14] CVE-2016-7054, "Chacha20/poly1305 heap-buffer-overflow," 2016. [Online]. Available: https://www.openssl.org/news/secadv/20161110.txt [accessed: 2017-07-26]

[15] H. Kario, "Tls test suite and fuzzer," 2015. [Online]. Available: https://github.com/tomato42/tlsfuzzer [accessed: 2017-07-26]