REDUCING HUMAN FACTORS IN SOFTWARE SECURITY ARCHITECTURES

Eric Bodden, Ben Hermann, Johannes Lerch and Mira Mezini

{eric.bodden, ben.hermann, johannes.lerch, mira.mezini}@ec-spride.de
EC SPRIDE, Technische Universität Darmstadt and Fraunhofer SIT Darmstadt

Abstract

In the recent past it has become clear that there are inherent problems with the security models of popular programming platforms such as Java, Android, and so forth. In this work we pinpoint sources of those problems, and explain the relative strengths and weaknesses of the security models for C/C++, Java, .NET, Android and JavaScript. As it turns out, many problems are caused by the fact that the models are so complex that they overstrain not only end-users but even expert developers. Out of this experience we argue that a new line of security models, based on object-capabilities, can help reduce the inherent complexity, preparing the ground for software that is secure by design.

Keywords: software security, security models, stack based access control, object capabilities

1 INTRODUCTION

Despite tremendous efforts from both the scientific community and the IT industry, the recent past has shown that current software platforms are hard to secure. Especially the most successful and thus most widely used platforms, such as Java and Android, have seen a lot of undesired attention through vulnerabilities that were discovered both by white-hat security companies as well as through the inspection of exploits that were already in the wild. In such situations it is easy to blame vendors such as Oracle and Google in the above cases, accusing them of negligence when it comes to assuring the security of those platforms. We, however, have performed a deep investigation on current security breaches, and have come to a different conclusion. While all currently popular software platforms have their very own security model, all those models share the unfortunate property that they impose a large number of explicit and implicit constraints on the way in which developers can write software in order for it to be secure, and on the way in which end-users can utilize the software securely. The sheer number of those constraints makes it very hard to reason about the security of software written for those platforms—for human developers, end users and for automated code-analysis tools.

In this work we pinpoint sources of this complexity for the security architectures of the common software platforms C/C++, Java, .NET, Android and JavaScript. Much of this complexity is not on purpose but is rather caused by the long history of these platforms. Software platforms strive to be backwards compatible, which often impedes the introduction of novel, potentially better security architectures. Another problem is that of performance: access control requires permission checks, which can slow down execution. Some platforms thus take shortcuts, unfortunately sometimes with drastic consequences for security and software maintenance. Last but not least, there is a problem with uninformed end users. Many platforms allow or even require end users to configure security settings themselves. However, many users are overburdened with this responsibility and grant capabilities to applications that better should have been denied. This problem is often aggravated by insecure defaults. By default, a malicious attacker can easily get access to sensitive resources on most platforms. The platform itself thus considers secure execution as a special case, which means that it must try to consistently apply access-control mechanisms to protect these resources, as an incomplete protection can easily yield a security vulnerability.

As a response to those findings we argue for a paradigm shift towards programming languages and platforms that follow a drastically different security architecture based on so-called *object capabilities* (OCaps for short). Architectures based on OCaps have the advantage that they follow the principle of least authority (POLA) [1] quite nicely, which, for good reasons, is popular and well known in the security community much beyond IT, security. With OCaps, a software platform is securely sealed by default. All security sensitive actions are encapsulated in special capability objects that applications can request from the platform. Importantly, every privilege can *only* be exercised by requesting the one and only capability that allows the application to exercise that very privilege. As we show, to allow a programming language to securely implement OCaps, it must not comprise a global namespace. Allowing reflective access poses a challenge and must also only be provided through a capability-based system. While being well aware of the fact that the introduction of such novel programming languages and platforms would mean a radical and revolutionary change for the software-development industry, we do believe that in the long term such a change is probably the only reliable avenue towards software systems that also in practice prove to be truly secure by design.

To summarize, this paper presents the following original contributions:

- a study of the security models of C/C++, Java, .NET, Android and JavaScript,
- a discussion of the inherent complexity of those models and how this complexity impacts software development, and
- an outlook towards novel, simpler to handle security models that follow the principle of least authority.

The remainder of this paper is organized as follows. Section 2 discusses the security models of current software platforms. Section 3 presents our outlook on novel models. We conclude in Section 4.

2 CURRENT SECURITY MODELS

We next give an overview of the security models of the currently most popular programming languages and platforms for enterprise, mobile and web applications, namely C/C++, Java, .NET, Android and JavaScript. For each platform we outline its security architecture and discuss its design tradeoffs, its strengths and current shortcomings.

2.1 C/C++

Applications written in C or C++ are commonly compiled directly into executable machine code. Opposed to so-called *managed code* known from platforms such as Java or .NET, machine code executes directly on the physical processor without any additional layers of protection (unless executed in a system virtual machine) and without much additional runtime support. C/C++ programs therefore essentially base their security entirely on support from the underlying operating system. The most important principle in this respect is that of strong process isolation: on virtually all known operating systems, each C/C++ application executes in its own process and is thus granted exclusive access to a given logical address space. It is the operating system's duty to ensure that other process cannot read from nor write to this space, hence securely protecting the application's data.

In result, security breaches mostly occur when an attacker manages (1) to hijack the process in question, or (2) the process contains errors in its programming logic that cause unsafe handling of process-external resources. Problems of the first kind often fall into the class of buffer-overflow attacks. Such attacks are made possible by the fact that many older C/C++ applications use unsafe library functions for string operations that are unchecked, i.e., which may write outside the bounds of a given array, thus overwriting arbitrary memory. An attacker can exploit this fact by overwriting specific memory locations known to reside on the execution stack with string characters that resemble executable code—the exploit's payload. Converting C/C++ programs to use checked string operations instead is not trivial. The problem is that in C/C++ an array does itself not carry any information about its size. Instead, an array is merely a pointer to its first element. In result, programmers must track the array length manually and consistently, which is cumbersome and error prone in itself.

Another possibility to hijack a process is through programming mistakes related to memory management. Through the lack of garbage collection, C/C++ imposes on the programmer the requirement to implement memory management manually. Such manual handling is known to cause different classes of bugs, including use-after-free or double-free vulnerabilities. In a use-after-free bug, the application code references a memory location that it actually previously cleared. Since the memory is cleared, it is no longer owned by the process, and can thus be overwritten by a malicious attacker. If then, for instance, the buggy process references the memory location as a function pointer, the process may inadvertently execute arbitrary attacker-defined code.

C/C++ also misses runtime checks for integer overflows. When an unsigned integer variable thus reaches its maximal possible value and is then incremented again, it falls back to the value 0. In a past attack on SSH, such a situation has lead an SSH security check to be switched off, as it was allocating an array of length 0 which should normally have been used to store and check connection information.[1]

So-called race conditions are problems of the second category, i.e., problems caused by errors in the programming logic. A race condition arises when the result of a computation can differ depending on the order in which processes are scheduled by the underlying operating system. In some situations, such "races" may break security. For instance, consider a situation where the C/C++ process closes a handle to a file (making the file accessible to the outside) and then changes the file's permissions to be world readable. An attacker could exploit the non-atomicity between those two operations to delete the file and replace it with a symbolic link, for instance to a password file `/etc/passwd`, tricking the original process into making this password file readable instead.

2.2 Java

Java was designed to be a programming language which on one hand should be easy to learn and on the other hand should allow one to write secure programs easier. Although Java is syntactically close to C/C++, it is quite different from a security perspective. In particular, Java is type safe and guarantees unforgeable references by forbidding pointer arithmetic or access to uninitialized memory. To reduce subtle mistakes often seen in C/C++ programs, Java supports automatic memory management and garbage collection, as well as automatic range checking of strings and arrays. A bytecode verifier, in conjunction with the Java Virtual Machine, guarantees type safety [2].

Java was one of the first language platforms that came with a built-in security model. Arising with the requirement that Java should be able to load and link code dynamically from arbitrary sources, the language designers had to deal with the problem that executed code cannot always be trusted. To address this issue, a sandbox was constructed, in which untrusted code is executed. This sandbox ensures that untrusted code is not able to perform actions that could do harm. The first versions of Java distinguished only between trusted and untrusted code, which are granted all permissions and no permissions respectively.

Java 1.2 introduced a more fine grained permission model. While before version 1.2 all local code was simply treated as trusted and any remote code as untrusted, the new model allowed

[1] Vulnerability Note VU#945216. SSH CRC32 attack detection code contains remote integer overflow
http: //www.kb.cert.org/vuls/id/945216

permissions to be assigned by highly customizable policies. The old model was acceptable, e.g., for applets, which are executed through a web browser. The code of the applet, which was loaded from a website, was treated as untrusted, while the system code of the locally installed Java library was trusted. When starting to consider more complex scenarios such as application servers, the granularity of the new model became important. Here the whole code is hosted locally, but not the whole code is being trusted, thus requiring more fine grained access control. For instance, a servlet used for the administration of the hosted web server may be granted the permission to write to the file system to allow for uploading other servlets, which themselves should not have that permission.

To check whether code is allowed to perform a security sensitive action the Java runtime library calls the check method of the `SecurityManager` class. If the calling code has the required permissions the check returns silently, otherwise it throws a `SecurityException`, which changes the control flow, thus preventing the execution of the sensitive action. While permission control became more fine grained with version 1.2, it still uses the same access-control techniques.

To avoid the possibility that untrusted code can ask trusted code to perform sensitive actions on its behalf (essentially a confused-deputy problem [3]), not only the immediate caller is being checked to have the required permissions, but the whole stack of callers. Moreover to allow for certain sensitive actions to be performed in predefined ways this simple principle must be weakened. For example, drawing to the screen is a security sensitive action, but should be possible if it is performed in an anticipated and controlled manner. Thus trusted code must have the possibility to control the permission check in such a way that it does not check its callers as long as the action to be performed is in expected boundaries. In Java this is done by invoking `AccessController.doPrivileged` while passing an object that defines what is to be executed.

In practice, we found that in Java permission checks are often limited to the immediate caller or to the n-th caller instead of the whole call stack, if there are n − 1 intermediate trusted callers through which a method is available. This implementation may be chosen out of performance consideration, as full stack walks can slow down a program's execution considerably. However, this design choice comes with the immediate danger that a new caller may be introduced in newer versions, which would result in the wrong caller being checked to have the required permissions. These limited permission checks therefore require newly introduced public APIs to perform permission checks on their own if they transitively call a sensitive action, as the existing checks have no effect in that case. It also implies that existing code cannot be easily changed. For example, the code of the Jana Runtime Library features comments such as *"does stack walk magic; do not refactor"*. Facing the problem, developers introduced additional specific checks that ensure only expected intermediate callers access a specific method. These checks use white-lists similar to the one shown in Listing 1.

```
/* Obtain the external caller class, when called from Lookup.<init> or
*  a first-level subroutine. */
private static Class<?> getCallerClassAtEntryPoint(boolean inSubroutine) {
  final int CALLER_DEPTH = 4;
    // Stack for the constructor entry point (inSubroutine=false):
    //   0: Reflection.getCC, 1: getCallerClassAtEntryPoint,
    //   2: Lookup.<init>, 3: MethodHandles.*, 4: caller
    // The stack is slightly different for a subroutine of a
    // Lookup.find* method:
    //   2: Lookup.*, 3: Lookup.find*.*, 4: caller
    // Note:  This should be the only use of getCallerClass in this file.
    assert(Reflection.getCallerClass(CALLER_DEPTH-2) == Lookup.class);
    assert(Reflection.getCallerClass(CALLER_DEPTH-1) ==
          (inSubroutine ? Lookup.class : MethodHandles.class));
    return Reflection.getCallerClass(CALLER_DEPTH);
}
```

Listing 1: Assumptions about stack layout in class java.lang.invoke.MethodHandles$Lookup

With the latest exploits of the Java API it becomes apparent that this security model may become hard to maintain even for professionals. Java 1.7 introduced a new class `ClassFinder`. An exploit filed under CVE-2012-4681[2] has been leveraging the public method `findClass` of this class to gain access to otherwise restricted classes. This method took a class name parameter and passed it without any check to a call to `Class.forName`. This parameter contains the name of a class to which `Class.forName` returns a reference. However, untrusted code should not be able to retrieve references to classes from restricted packages such as those with the prefix sun, as many of them provide access to security sensitive actions. To prevent untrusted code to get those references, `Class.forName` checks if the immediate caller has the required permissions. As `ClassFinder.findClass` – its immediate caller as can be seen in Figure 1 – is part of the trusted code it has these permissions and leaks out the class reference to the untrusted code.

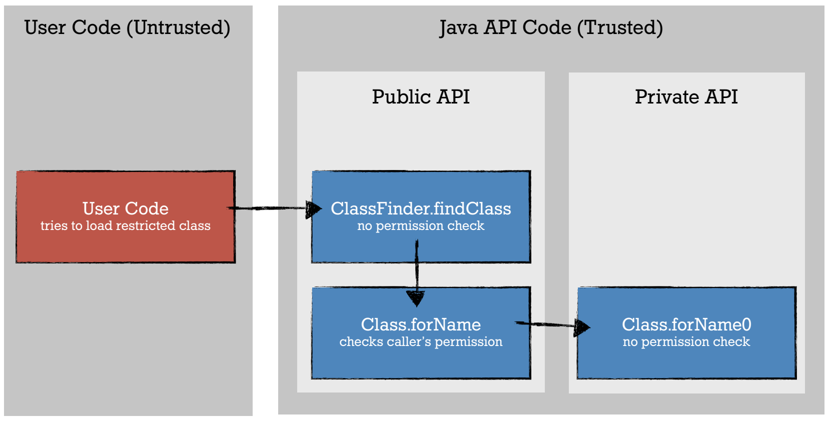


Figure 1: Call structure for the exploit on ClassFinder.findClass (in the Oracle JDK 7 Update 10)

Using this leak, the exploit was able to gain access to a security sensitive class in a sun package. Here another problem becomes visible: Java's global namespace. If namespaces would be strictly separate, untrusted code would not be able to even use restricted code.

In summary, everything in the Java library that provides access to security sensitive actions must perform permission checks on the SecurityManager. Therefore, Java follows a design where by default everything is allowed, and must thus be revoked individually for unprivileged code by performing security checks.

[2] Common Vulnerabilities and Exposures—http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4681

2.3 .NET Framework

Microsoft's .NET Framework was first released in 2002. It provides a language-neutral infrastructure for application development and execution. Code in various languages can be compiled to an intermediate language (IL)—much like Java bytecode—to then be executed via a just-in-time compiler that translates the instructions to the respective machine code. Most similar to Java bytecode IL code is validated and verified by the runtime before the execution. First the validation step checks for proper IL instructions and the verification step guarantees the type safety of the executed program. During runtime permissions are checked against policies by the Code Access Security (CAS) model in the runtime environment. Thus, the security model is not depending on a specific language but rather on the platform.

Permissions can be requested by assemblies—the means of packaging applications or libraries in .NET—and either granted or denied by policies. They can be organized hierarchically to allow flexibility. Assemblies can be signed to form a "cryptographically sealed" namespace. But as these signatures are only a means of self-identification and as they furthermore cannot be revoked, they only provide a cryptographic proof of the integrity of the assembly. Signing assemblies creates a "cryptographically sealed" namespace that can be used as evidence in policies, and to isolate or secure specific assemblies.

To check specific permission demands against existing policies, the runtime performs stack walks. In contrast to Java, only full stack walks are performed. Exceptions to this rule are only made when a callee explicitly vouches for its callers, through a mechanism called LinkDemand.

Nevertheless, the .NET Framework has been struck with exploits quite similar to those Java is facing. Especially issues like CVE-2013-0004[3], CVE-2012-4777[4] or CVE-2012-1895[5] show that also the CLR version of stack-based access control seems hard to maintain when adding new functionality to the platform.

2.4 Android

Android is one of the most popular open-source software stacks for mobile phones. It is developed by the Open Handset Alliance, which includes Google, T-Mobile, EBay and other well-known companies. Android apps are programmed in Java but interestingly do not at all use the Java security model. In fact, Android apps execute with the Java security manager disabled. Instead, Android bases its security entirely on process isolation, with the same benefits and drawbacks discussed for C++ and C. If an app is written exclusively in Java, it enjoys the memory safety of that language, making impossible buffer overflows and bugs related to manual memory management. Android apps can include natively compiled C/C++ code, however, which then *is* prone to those problems.

Due to the strong process isolation, apps execute in different address spaces, making it impossible for one app to obtain a direct reference to the memory of another—a property much different, for instance, from servlets in Java. In result, the programming model is largely simplified, and the Android standard library yields a comparatively small attack surface. As a drawback, Android's access-control infrastructure is quite limited. An app is granted a given set of permissions at installation time, causing its process to become equipped with the appropriate capabilities when it is started. There is no possibility for more fine-grained access control. Further, due to the isolated address spaces, Android apps can only exchange data through inter-process communication, which involves (a lightweight form of) serialization.

[3] Common Vulnerabilities and Exposures—http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0004
[4] Common Vulnerabilities and Exposures—http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4777
[5] Common Vulnerabilities and Exposures—http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1895

While giving the advantage that one app cannot possibly leak a memory reference to another, this mechanism is less convenient and efficient than direct in-memory access as in Java. The Android platform is built on an embedded Linux system and its libraries. Each app is executed within the context of an own virtual “user” giving the app the opportunity to store privately files into the otherwise shared file system.

As a result of these measures, existing security problems regard rather end users than developers. Many problems are caused by the permission model. While Java leaves the definition of security policies to experts maintaining the server or browser, Android asks the end user to decide on whether a certain set of permissions should be granted to an app at installation time. Many users find such decisions hard to make, also because it is hard to judge for a user whether an app indeed requires a certain permission to conduct its business operations or whether the permission might in fact just be required to conduct malicious activities. Moreover, the permission system is quite coarse grain. An attacker could, for instance, instrument an existing social-networking app with malware. A user will certainly grant the “internet” permission to this app, with good cause. The malware code could exploit this permission, though, to send off any and many apps include. Unlike in Java, such frameworks cannot run in a restricted access-control context. Instead they run with the same credentials as the app itself, and thus with the same potential access to private data. In conclusion we can see that the deviation of Java from its old sandboxing model was not without good cause: Android shows that a more advanced and fine-grained permission control can be desirable.

2.5 JavaScript

JavaScript is traditionally used to add dynamic elements to websites, therefore executed inside web browsers. It is a scripting language used in a closed environment intended not to be able to access anything outside the browser. Exceptions are access to cookies, sending requests to servers, which typically are limited to the web server from which the code itself was initially loaded. This access is provided by native browser methods that underlie strict limitations. Having these strict boundaries of what is allowed, the JavaScript language itself had—from a historically point of view—never the requirement of a security model. This changed in recent times with websites that serve as a platform for applications provided by different developers to enrich the the originally offered services (e.g. iGoogle, Facebook, Yahoo! Application Platform). These applications are delivered together with the main website, as a so-called mashup. As JavaScript does not come with a security model, any of those embedded applications can interfere with the main website or each other. This can become dangerous if the code originates from untrusted sources.

Platform providers have thus developed a variety of add-on security models, e.g., Google Caja, Facebook FBJS, or Yahoo! ADsafe. Basically, these models try to separate applications from the main website and also from each other. Access to data or the main website is provided through predefined access objects, so-called capabilities. To ensure that security sensitive actions are only possible through capabilities that are explicitly passed, there must be no other way to access them. In other words, the language must enforce strict encapsulation without global variables etc. The original JavaScript language is too rich to provide such guarantees, which is why the security models enforce usage of a subset of JavaScript, typically through checks on the web server.

Recent investigations have shown that security models that were already in use for quite a while still miss possibilities that allow breaking the encapsulation [4], [5]. Most of the issues were really specific and not obvious, like using the prototype mechanism to create communication channels or simply using references to sensitive variables passed indirectly as attributes of other objects. To conclude, JavaScript is interesting in the sense that its dynamic nature allows researchers and practitioners to add security support after the fact, at least if certain language features are disabled.

3 TOWARDS SIMPLER SECURITY MODELS

In the preceding sections we have shown that existing security models are complex to handle, probably too complex for most developers and end users. In the remainder we discuss why we believe that novel security models based on object-capabilities might solve some of those problems.

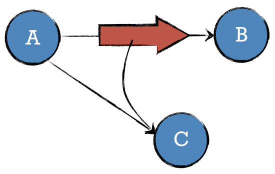
3.1 The Object-Capability Paradigm

In an object-oriented system objects communicate with each other through messages that an object can send if it has a reference to another object it would like to reach. Current object-oriented programming models such as Java further provide access to "static" members that are reachable without a direct reference, through a global namespace. Static methods are popular for supporting convenient ways to execute specific algorithms on provided data. These programming models usually also provide reflective capabilities to obtain access to the internal structure of objects.

In the object-capability model (OCaps) [6], a program's reference graph becomes its access graph. Objects may only interact by messages passed along those references, much like in any object-oriented system. To allow such a system to be secure, references must be unforgeable and protected. Thus, programming-language features such as pointer arithmetic or reflection conflict with the object- capability model. The same holds for static fields as available in languages such as Java, as they also would allow for interaction outside the reference graph. As messages can carry references, there are only four ways for an object to receive a reference to another object:

initial state: The initial program state is granting a reference from object A to object B.
parenthood: When A creates an object B, A is the sole owner of that reference to B.
endowment: While creating B, object A endows object B with a subset of its references.
introduction: When A sends a message to object B it can add a reference to object C in the message if it had that reference.

The last situation, introduction, is depicted to the right. As references in an object-capability system must be explicitly passed through messages, it becomes much easier to analyze programs for security properties such as information flow. Also, stack-based access control checks can be reduced to the time of object creation and—depending on the architecture—when leaving privileged areas. This makes the security model more efficient during runtime.



Modern security architectures already implements parts of the object-capability paradigm. For instance the implementation of the `java.lang.invoke` API in Java 7 is clearly inspired by OCaps, as it performs access-control checks only when a method handle is created but not when it is invoked. This is in contrast to the older (and slower) API `java.lang.reflect`. The object-capability model is also central to the language design of the language E [7]. Following E's example, the research community built many object-capability aware subsets of common languages, e.g., Joe-E [8] as a subset of Java, or Caja [9] as a subset of JavaScript. Although these languages mostly serve as research prototypes, they demonstrate the potential this security model has.

The object-capability model can also serve a guiding framework when assessing the security of pre- existing software artifacts such as libraries. Violations of the model, like the use of native code (theoretically making references forgeable), reflection (possible violation of encapsulation) and accesses to static members (ambient resources) can be reported so that developers can make an educated decision on the use of libraries, thus making systems possibly more resilient.

3.2 Type systems and history-based access control

A type system, in general, associates metadata with values that a program computes. This metadata is called a *type*, and is typically denoted through a type annotation on a variable holding that value. In result, the type describes properties of all values the annotated variable could hold during its lifetime. As a simple example, the declaration `int i` in Java describes the fact that the variable `i` can only hold integer-typed values. A type checker uses a type system to find type errors, i.e., programming errors that contradict the program's assumptions expressed through types. In Java, for instance, a type checker would raise an error on any attempt to cast `i` to a pointer variable, as `i` does not describe a pointer. (C's type system is unsound and would, in fact, allow such casts.)

Security Type Systems Over the past years, researchers have invented security type systems that encode security assumptions through additional metadata. Jif [10], for instance, is a type system for secure information flow, in which variables can be annotated with security levels such as *high* and *low*. High levels are typically assigned to variables carrying secret or private data, while variables used for output purposes are typed low. The type checker assures that high variables cannot be assigned to low ones, hence keeping secret data secret by design. One problem with type systems is that they are generally context-insensitive. This property originates from the fact that type systems assign to variables metadata that is valid in *all contexts* through which the variable is accessed. In real-world programs, however, the properties of the values that a variable refers to may change depending on the context. For instance, the same variable might sometimes carry *high* and sometimes *low* data. In result, most real-world programs, even if secure, would actually violate existing security type systems.

Dependent Classes Dependent classes [11] were introduced as a means to express variability in software systems. In this system, classes may depend on an arbitrary number of objects. For example, a class A that depends on an object b is of a different type than a class A that depends on an object c, even though b and c may themselves be instances of the same class. A type-checking algorithm can thus distinguish these types from each other, and can therefore guarantee separation for the respective types and objects. Exactly this property is an interesting guarantee for the promotion of secure design practices. For instance, a framework provider may choose to bind security critical information to specific targets, e.g., URLs. When sending information to these targets, the type checker will only accept parameters that match those dependent targets, making security effectively a compile-time or verification-time check.

Singularity Singularity [12] is an experimental operating system developed by Microsoft Research. It is based on the general principle that virtually all code, including that of the operating system itself, is written in type-safe languages, and executed in managed code, supporting garbage collection etc. Quite interestingly, such a design avoids all pointer-related low-level security problems known from C/C++. In result, it is actually safe and even desired to allow processes to share memory regions—an important departure from the process-isolation model known from virtually all other operating systems. Memory sharing between processes has big advantages when it comes to performance. With regular operating systems, for instance, a document arriving over the network and being forwarded to the printer typically has to be copied over and over again through different memory making up the layered architecture that those operating systems provide. In singularity, the data itself can actually stay at one place, and the operating system can instead simply pass pointers between those layers. Such a platform design is only secure due to the language safety provided by modern type systems. In particular, the type system must enforce that references cannot be forged.

History-based access control As we explained, many of the recent security problems experienced with Java and similar platforms are due to problems with stack-based access control. Abadi and Fournet [13] have already identified stack-based access control as problematic ten years ago, and have proposed a history-based access-control scheme as an alternative. In such a scheme, permission checks do not only include the privileges of all code currently on the execution stack but rather of all code that was executed during the history of the program observed so far. While certainly more secure, such a scheme seems also more restrictive: if a trusted program's execution ever calls untrusted code (which has no permissions), then this will, by default, void all of the caller's permissions, even if the untrusted code does nothing whatsoever. Ideally, an access-control scheme would be able to determine any possible detrimental effects that this code may have on the execution of the trusted program. However, this would generally involve information-flow analysis which is generally hard to scale and to reason about.

4 CONCLUSION

In this work we have surveyed and contrasted the security models of existing programming languages and platforms. As we identified, there is an inherent trade-off between simplicity and flexibility. Simple models tend to provide stronger security guarantees and are less likely to provoke implementation bugs. On the other hand, such simple models tend to be less flexible than desired, for instance forcing users to grant more permissions than strictly necessary for an application's execution. More flexible models come at the cost of increased usage and implementation complexity. Such models typically induce code that requires access-control checks at many places, thus yielding a comparatively large potential for programming errors, a human factor that is not to be under-estimated. Also, more flexible access-control policies are harder to define correctly by end users. Questionable is the technique of taking shortcuts on access-control checks such as the degraded stack-based access-control implemented in some parts of the Java runtime library. While yielding more efficient code, the technique causes tremendous problems with respect to code maintenance.

We conclude that further research on the topic of secure programming models is required. Novel programming models, for instance based on object capabilities, have the potential to simplify the enforcement of access-control checks. While also such novel security models need to be accompanied by proper guidance and training, they have the potential to yield software that is more secure by design, because opposed to existing approaches those models can provide secure defaults.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the DFG within the project RUNSECURE.

REFERENCES

[1] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[2] Li Gong. Java security architecture (JDK 1.2). Technical report, 1998.

[3] Norm Hardy. The confused deputy (or why capabilities might have been invented). ACM SIGOPS Operating Systems Review, 22(4):36–38, 1988.

[4] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In Michael Backes and Peng Ning, editors, ESORICS, volume 5789 of Lecture Notes in Computer Science, pages 505–522. Springer, 2009.

[5] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In IEEE Symposium on Security and Privacy, pages 125–140.

IEEE Computer Society, 2010.

[6] Mark Samuel Miller and Jonathan S Shapiro. Paradigm regained: Abstraction mechanisms for access control. In Advances in Computing Science–ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation, pages 224–242. Springer, 2003.

[7] Mark Samuel Miller and Jonathan S Shapiro. Robust composition: towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University, 2006.

[8] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of Java. In Network and Distributed Systems Symposium. Internet Society, 2010.

[9] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. Technical report, Tech. Rep., Google, Inc, 2008.

[10] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. Selected Areas in Communications, IEEE Journal on, 21(1):5–19, 2003.

[11] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In ACM SIGPLAN Notices, volume 42, pages 133–152. ACM, 2007.

[12] Galen Hunt, James Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[13] Martın Abadi and Cédric Fournet. Access control based on execution history. In Proc. 10th Annual Network and Distributed System Security Symposium, 2003.