

# Programming Languages and Systems Security

Programming languages communicate ideas between humans and machines. As a designer's means of expression, they underlay any information system's definition and implementation, and are thus fundamental to system design and construction.

CHRISTIAN  
SKALKA  
University of  
Vermont

Recognizing this, and recognizing security's importance in modern information systems, programming language researchers have made security a primary concern in language design and implementation. In this installment of Secure Systems, I briefly survey the current language-based security technology, particularly as it affects secure system design.

## **Secure foundations for systems design**

Systems designers typically develop secure systems architectures at higher levels than the programming-language level. While specification languages such as Unified Modeling Language (UML) provide a way to express higher levels of system abstraction, the principal contribution of programming languages to systems security is low- and intermediate-level support for high-level abstractions.

In other words, security policies that specify expectations and requirements of system usage are born in the designer's imagination, whereas programming-language security supports a faithful and robust implementation of these ideas in a variety of ways. For one, well-defined and reliable program execution prevents attackers from circum-

venting security policies by exploiting weaknesses in language models. For another, many modern languages provide primitive features for specifying and enforcing security policies, such as the ability to modify access-control lists and implement access checks. Also, language compilers and runtime systems are a natural setting for imposing security controls on program execution. In short, modern languages give systems designers confidence in their underlying programming models' safety and reliability, and a flexible and expressive way to precisely specify and rigorously enforce high-level security policies.

## **Language safety and type safety**

*Language safety* is a generalization of common notions such as *type safety* and *memory safety*. In essence, it's the assurance that language implementations—including compilers, type analyses, and runtime systems—enforce a program's intended semantics. Type analysis ensures that programmers use language features in a manner consistent with their definitions—for example, functions defined on integers shouldn't be applied to array values. Type safety is a formal guarantee that static (compile-time) type analysis enforces safe

program behavior, although such behavior can also be enforced by dynamic (runtime) checks, or a combination of the two. Computer professionals frequently misidentify type safety and language safety, in large part because type systems are a relatively lightweight and inexpensive static program analysis for enforcing language safety. Indeed, strong static-type analysis enforcing some form of type safety is fundamental to modern programming practice, and is present in Java and C#. Type safety is a common theme in language-based security; as languages subsume and clarify notions of security, the meaning of language safety evolves, so type systems and safety guarantees must evolve as well.

Language safety is relevant to systems security because many attacks exploit vulnerabilities in language design and implementations, circumventing system security by hijacking the program execution model. Perhaps the most well-known example of such an exploit is the buffer overflow attack.<sup>1</sup> The vulnerability is native to C and C++, in which programmers—via, for example, the `strcpy` function—can copy data to allocated buffers without ensuring that the data is no larger than the buffer. This lets buffers “overflow,” so that the overflowing data overwrites unallocated memory. If the buffer is allocated on the call stack, malicious code can exploit buffer overflow to overwrite the stack-frame region that stores the function return address. An attacker can replace the “true” return address with a reference to code that he or she chooses, hijacking the language execution model to run arbitrary

code with all the exploited program's privileges.

Buffer overflow attacks have a formidable history dating back to the Morris worm (1988) and contributing to 24 out of 44 CERT advisories published between 1997 and 1999. Given this exploit's prevalence and destructive potential, it's somewhat surprising that not only is the attack well understood, but that several completely effective defenses against it exist, because current day black-hats continue to employ "stack-smashing" with seeming impunity. At the simplest level, C and C++ provide an alternate `strncpy` function for more careful memory management. However, this solution doesn't protect against programmer error or vulnerable legacy code.

Programming-language safety allows a more principled defense against buffer overflow and other exploits. From the bottom up, safe languages are designed to ensure that language implementations enforce intended program semantics during execution. Various modern safe languages exist—including OCaml, Java, and C#—that employ a combination of static analyses (such as type inference in OCaml) and dynamic checks (such as cast checking in Java) to enforce safety.

Language researchers have even extended type-safe memory management to the intermediate and assembly language levels,<sup>2</sup> statically ensuring efficient memory safety for any higher-level object language. In safe languages, exploits such as buffer overflows aren't just preventable through better programming practice—they're literally impossible to program, as an effect of safety. Secure systems engineering is a common application of safe-language technology; for example, researchers have used PLT Scheme, a modern safe dialect of Lisp, to write an advanced Web server<sup>3</sup> that, unlike Web servers written in C such as Apache, isn't subject to buffer overflow exploits.

Of course, a cultural affinity for C

and C++ persists due to ingrained programmer bias and legacy code, as well as the languages' many real benefits, such as speed and powerful low-level control. The CCured system addresses this reality, comprising a type analysis and runtime checks for "retrofitting" safety properties in legacy C code.<sup>4</sup> This research has also led to the development of Cyclone, a safe C dialect that can prevent common C exploits such as buffer overflow and format-string attacks.

### Language-based security abstractions

In addition to traditional support for defining functions and manipulating data types, advanced programming abstractions such as threads and distributed message passing have become common in modern languages. Associated language features give programmers powerful, flexible control over various resources. Recent language-based security research and development seeks to add security behaviors to language execution models and features for programming security policies to language syntax. As others have observed,<sup>5</sup> providing language-based control over system security promotes robustness and flexibility because it gives programmers direct and fine-grained control over security policy.

Both the Java Virtual Machine (JVM) and .NET Common Language Runtime (CLR) provide language-based support for specifying

code with authorization levels via code-signing techniques and access-control lists, access-control decisions can be predicated on active code authorization levels—literally by inspecting the call stack to discern owners of preceding function activations. This kind of access control can only be realized in a language implementation, and provides distinct benefits such as defense against man-in-the-middle attacks. Stack-inspection access control contributes to JVM and CLR protection mechanisms. In general, it provides language-based policy enforcement for extending local systems with nonlocal code; for example, we can use stack-inspection access control to disallow applets from communicating with URLs other than their source URL. Although the JVM and CLR both implement stack-inspection access control with a dynamic check, recent research has demonstrated that novel type systems can statically enforce this security model,<sup>6</sup> rendering dynamic checks' significant overhead unnecessary.

An alternative to access control is capability-based security. In this model, possession of an unforgeable reference to some sensitive object is sufficient to use that object, and reference distribution is at the heart of security policy. The capability-based protection mechanism in the Extremely Reliable Operating System (EROS) can enforce traditional OS protection schemes, as well as more sophisticated properties such as the Bell-Lapadula \*- property. Several

## Programming-language safety allows a more principled defense against buffer overflow and other exploits.

expressive access-control policies. The *stack-inspection* flavor of access control provided in these settings is predicated on program-execution contexts.<sup>6</sup> In particular, by associating

languages and extensions, such as E (www.erights.org), Java confinement types,<sup>7</sup> and the Luna system,<sup>8</sup> bring control of capability-based security to the language level, letting program-

mers control protection mechanisms often found at the OS level.

Language researchers are also interested in security abstractions that allow the specification and enforcement of protocols for communication and resource usage—temporal properties that rely on well-formed action sequences. For example, well-mannered programs should open files before they're written to, establish Secure Sockets Layer connections via strict protocol before packets are sent, and so on.

Although expressive model-checking techniques have long helped ensure temporal properties of system specification, researchers are now proposing modern type systems to specify and enforce temporal program logics efficiently and declaratively at the language level.<sup>9</sup> The Vault language<sup>10</sup> uses a static type system endowed with “type guards” that enforce a general class of temporal properties, which programmers can specialize to enforce domain-specific usage protocols. In particular, experimental Vault programming of the interface between the Windows 2000 kernel and its device drivers allows static specification and enforcement of protocols governing I/O request packets (IRPs), I/O request completion routines, thread coordination, and interrupt levels.

### **Controlled execution environments**

Programming languages provide access to local computing resources, such as CPU, memory, and files, and systems attacks often abuse these resources. Even in a safe programming environment, languages can provide an inroad for denial-of-service (DoS) attacks, for example, by allowing access to such resources. However, given that safe programs execute within the model that language implementations provide, it follows that implementations can effectively provide a restricted execution environment in case resource

use should be moderated. Although some sources refer to this as access control, here I distinguish between these concepts, designating access control a security abstraction available to programmers, not a modification of the underlying program-execution model.

*Sandboxing* is a general term for restricting a process's access to system resources, and is realized in systems in several ways. Often implemented at the OS level, as in TRON for Unix, we can achieve sandboxing in both the Java and the CLR language platforms by combining various lower-level language safety and security features—for example, type safety and stack-inspection access control. The class loader also contributes to Java sandboxing via namespace management, demonstrating how we can selectively modify language semantics for security. For example, the JVM prevents untrusted programs such as applets from loading classes with the same names as existing libraries to protect local systems against spoofing attacks. Trusted code, on the other hand, receives less restricted class loading, hence it has more powerful control over its namespace, even allowing redefinition of the class loader itself. The Luna system has pressed this idea even further,<sup>16</sup> extending Java's type system with support for process separation via namespace and capability management, a protection traditionally obtained at the OS level using virtual memory.

Java is a general-purpose programming language, but special-purpose languages exist that also employ computational restrictions for systems security. The Packet Language for Active Networks (PLAN)<sup>11</sup> disallows programming loops of nonfixed length and the use of unrestricted recursion—the programmer must hard-code all looping and recursive bounds. Because this restriction ensures program termination, we can use PLAN for program-

ming active networks, but not for DoS attacks because termination ensures that CPU and memory won't be consumed out of system bounds. In this way, PLAN resolves a problem intimately associated with active networks, which is their vulnerability to DoS attacks.

**M**odern research and development has produced various language-level supports for secure systems design. Safe languages provide a flexible and reliable foundation on which to build. Language-based security abstractions provide systems programmers with an effective means of defining and enforcing security models. Controlled language-execution models can impose fine-grained and powerful restrictions on code at varying levels of trust.

Arguably, the most popular modern general-purpose languages, Java and C#, bear witness to the importance of programming-language safety and security, being both safe and endowed with sophisticated security models. Research into these topics is ongoing, but perhaps the most important current task is the integration of modern language security technologies—much more efficient and effective than past technologies—with realistic systems design. □

### **Acknowledgments**

*The author thanks Michael Hicks for his insights into languages and systems.*

### **References**

1. Aleph One, “Smashing the Stack for Fun and Profit,” *Phrack*, vol. 7, no. 49, 1996.
2. K. Crary and G. Morrisett, “Type Structure for Low-Level Programming Languages,” *Proc. Int'l Colloquium Automata, Languages, and Programming*, LNCS 1644, Springer-Verlag, 1999, pp. 40–54.
3. P.T. Graunke et al., “Programming the Web with High-Level Pro-

- gramming Languages,” *Proc. 10th European Symp. Programming Languages and Systems (ESOP 01)*, LNCS 2028, Springer-Verlag, 2001, pp. 122–136.
4. G.C. Necula, S. McPeak, and W. Weimer, “CCured: Type-Safe Retrofitting of Legacy Code,” *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 02)*, ACM Press, 2002, pp. 128–139.
  5. C. Hawblitzel and T. von Eicken, *A Case for Language-Based Protection*, tech. report TR98-1670, Dept. Computer Science, Cornell Univ., 1998.
  6. F. Pottier, C. Skalka, and S. Smith, “A Systematic Approach to Static Access Control,” *Proc. 10th European Symp. Programming (ESOP 01)*, LNCS 2028, Springer-Verlag, 2001, pp. 30–45.
  7. J. Vitek and B. Bokowski, “Confined Types in Java,” *Software—Practice and Experience*, vol. 31, no. 6, 2001, pp. 507–532.
  8. C. Hawblitzel and T. von Eicken, “Luna: A Flexible Java Protection System,” *SIGOPS Operating System Rev.*, vol. 36, 2002, pp. 391–403.
  9. C. Skalka and S. Smith, “History Effects and Verification,” *Proc. Asian Programming Languages Symp.*, LNCS 3302, Springer-Verlag, 2004, pp. 107–128.
  10. R. DeLine and M. Fahndrich, “Enforcing High-Level Protocols in Low-Level Software,” *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 2001, pp. 59–69.
  11. M.W. Hicks and A.D. Keromytis, “A Secure PLAN,” *Proc. 1st Int’l Working Conf. Active Networks*, LNCS 1653, Springer-Verlag, 1999, pp. 307–314.

*Christian Skalka is an assistant professor at the University of Vermont. His research interests include programming language-based security and type theory. He received a PhD in computer science from the Johns Hopkins University. Contact him at [skalka@cs.uvm.edu](mailto:skalka@cs.uvm.edu).*



# IEEE distributed systems ONLINE

Expert-authored articles and resources

*IEEE Distributed Systems Online* brings you peer-reviewed articles, detailed tutorials, expert-managed topic areas, and diverse departments covering the latest news and developments in this fast-growing field.

Log on <http://dsonline.computer.org> for *free access* to topic areas on

- Security
- Grid Computing
- Mobile & Pervasive
- Distributed Agents
- Middleware
- Parallel Processing
- Web Systems
- Real Time & Embedded
- Dependable Systems
- Cluster Computing
- Distributed Multimedia
- Distributed Databases
- Collaborative Computing
- Operating Systems
- Peer-to-Peer

<http://dsonline.computer.org>

To receive regular updates, email [dsonline@computer.org](mailto:dsonline@computer.org)