Implementing Security Via Modern Programming Languages

Jaime Niño
Computer Science department
University of New Orleans
New Orleans, LA 70148
jaime@cs.uno.edu

Abstract

Security topics in all areas are a pressing need for Computer Science instructors. This paper provides a survey of security features in modern programming languages. We present the role that type safety and capabilities provide for the building of secure systems, and how language systems allow designers to model security issues that once were part-and-parcel of Operating Systems, or that can not be modeled by such.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: language features.

General Terms

Languages, Security.

Keywords

Access control, information flow, type safety, capability.

1. Introduction

Modern programming languages promote system security by providing features that deliver robust implementations of security policies in a variety of ways. First, many languages provide features for specifying and enforcing security policies, such as the ability to define and modify access-control lists, and analyze data control flow. Second, well-defined and reliable implementation of access checks, and management of program execution prevents the circumventing of security policies by attackers by exploiting weaknesses. Also, language compilers and runtime systems are a natural setting for imposing security checks on program as well as controls on its execution. Thus, modern languages give confidence to systems designers in their underlying programming models' safety, reliability, as well as provide flexible and expressive ways to precisely specify and rigorously enforce security policies. In short, modern languages are now more than just tools for programming computations.

The goal of this paper is to provide instructors with a small but comprehensive survey of security in programming languages, not secure languages per-se, for the course in Programming Languages as well as other courses. Reasons for this topic include the following:

- Proactive approach to security: Building secure systems must start with the use of safe programming languages.
- Material for instructors not expert in security and Programming Languages, as it transpired in a BoF session. [17]
- A survey of popular books in Computer Security [1, 3, 28, 26] as well as in Programming Languages [22, 24, 27] indicated that little or no material is dedicated to programming languages and security.

We start by introducing basic security concepts in sections 1, and 2. In the remaining of the paper we discuss security via language mechanisms. Only a basic understanding of security implementation strategies, as found via common operating systems, is required from the reader.

2. Basic Security Definitions

Security is often discussed in terms of security policies, principals, objects, and protection domains. [1, 3, 26]

A *security policy* states what is and what is not allowed by the system. *Principals* are the entities whose actions must be checked for legality according to a security policy. Principals invoke operations on objects. An *object* may be a data value, a file, a hardware device, and in general a system resource. The context within which a principal executes is called as a *protection domain*; cross-domain operations must be authorized by a security policy. The *Trusted Computing Base, (TCB),* is the totality of protection mechanisms, including hardware, firmware, and software, responsible for enforcing a security policy.

3. Principles And Minimal Safety Policy

Development in language design for security has been guided by two classic computer security principles [23]:

- **TCB**: Assurance that enforcement mechanism behaves as intended is greatest when it is *small*, *simple* and *verifiable*.
- ***Principle of Least Privilege (POLP).*** Throughout execution, each principal should be accorded the minimum access needed to accomplish its task.

For a better understanding of the inroads made by programming language design for the development of secure system let's begin by decomposing the system security requirements into its two major concerns: *access control* and *information flow*. Authorization or access control answers the question of which principals are permitted to access which resources. Information flow specifies the permissible consequences of properly authorized access to a resource.

Enforcement of authorization has usually been the venue of the OS (kernel, reference monitor, privilege library). Specifically, protection mechanisms are provided in hardware for memory protection (virtual memory), user/supervisor execution levels, and traps to switch levels. Coupled to memory protection, system security policy is implemented at the OS level via an access control system, discretionary or mandatory, with a runtime monitor checking for the reading and writing of sensitive data by authorized principals. The accesses are verified by a search on the access control list. An access control list contains, per object, each principal that can access it and the operations that the said principal can perform on the object. [3]

/SIGCSE'09/, March 4-7, 2009, Chattanooga, Tennessee, USA.

At the very minimum, any security policy should guarantee the following fundamental safety properties:

- *Control flow safety.* Program should never execute a jump or call to a random location, but only addresses within its own code containing valid instructions. All calls should be to valid function entry points and all return to the location from which the function was called.
- *Memory safety.* The program should not access random places in memory, but only valid locations in its own static data segment, live system heap memory explicitly allocated to it, and valid stack frames.
- *Stack safety.* The runtime stack should be preserved across function calls. Minor modications near the top of the stack are allowed, as is tail recursion elimination.

4. Security Implementation Via Software

Security has been turning into a software issue as the mechanisms used to implement security policies are cheaper and more flexible and portable in software than in hardware. Moreover, systems are particularly vulnerable to security attacks at the *application level* (e.g., Lovebug and Melissa viruses). However, application-level attacks operate at a higher-level and circumvent OS security mechanisms. In addition, implementing security policies at the programming language level has been advocated for several reasons [9, 29]

- Language semantics can help reasoning about program behavior and thus prove security properties.
- Type systems and static analysis algorithms can reduce security run-time cost.
- Protection domains can be made lightweight and allow fine-grained interactions.
- Language-based protection allows specification of access rights with more precision than virtual-memory based mechanisms. Via hardware protection only locations are protected, not the value or what can be done with it.
- Via language-based protection, calls across protection domain boundaries could be as cheap as simple function calls, enabling as much communication between components as desired without performance drawbacks as found via hardware protection context switches.

It must be emphasized that via the OS security implementation is checked at run-time and it might be valid for certain executions, while language-based security implementation can be checked statically for the most part, and its correctness will be so for all executions. Besides, there are security policies whose enforcement cannot be monitored by a single execution, including resource availability and information flow.

Also due to the precision of policy specifications allowed by programming languages, they provide a natural setting for the implementation of POLP, as different computations usually require different degrees of authority.

On the other hand, advantages of hardware based protection over software-based we find the following factors. [29]

- *Size of the TCB*. All software used to implement security must be added to the TCB and should be small and easy to prove its correctness. (An unlikely event?)
- *Single language restriction.* Traditionally systems using language-based protection require all code to be written in the same language. This should no longer be an issue; witness the existence of the .NET languages and their Common Language Runtime as a model where different languages compile to the same VM bytecode.
- *Garbage collection.* Type-safe systems require automated memory allocation which places a performance hit and adds to the size of the TCB.
- *Revocation of access rights to data*. This is easily done at the virtual memory hardware by changing the page tables. In type-based systems a direct access to data could be done via invalidation of references, a hard problem due to aliasing [29], or by adding a level of indirection to the data.
- *Regular code performance*. This is the possibly unavoidable cost of security checks at run-time.

5. Security and Type Safety

Language implementation security starts with the use of a type safe programming language. While type safety is not the same thing as security it is an essential foundation for the latter. The fundamental characteristic guaranteed by type safety (strong typing) is *type soundness*: well-typed programs do not go wrong. As witness in languages such as C/C++, violations of type safety constitute serious security threats.

In contrast to address-based protection, as supported via the OS, the address at which a piece of data resides is irrelevant in language-based protection. Only the type of the data with its operations are relevant. This turns out to be a beneficial feature of security via types, as it allows finer control on accessibility of data contrary to the virtual page sizes.

A key issue of language based protection is ensuring that type safety and access control are enforced. It's a well-known fact that type enforcement can be done statically or dynamically. We must be aware that via dynamic type checking, errors may leak information. Further, in the presence of type safety, the minimal safety requirements, listed in section 3, are subsumed by it. But, a requirement to be type-safe, a language must have garbage collection or otherwise restrict the allocation and deallocation of memory.

Examples of type safe languages include Ada, ML, Java, Haskell, Lisp, Scheme, Modula-3 among others.

Language safety is a generalization of the notions of 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. Therefore, language safety can add a large footprint to the TCB. In particular the type checker must be verified. Moreover, it must be verified that the type information has been correctly *preserved* in the object code, quite a task on the face of the size and complexity of modern languages and compilers, and the object code with no type information.

A method to reduce this complexity and size is to map the high level type information to a lower (read simpler typed) level language that transforms the code while preserving the type information [20]. An instance of this approach is found in Java. It compiles to JVM bytecode and the proof of type preservation is produced by the bytecode verifier phase during class loading. Same for the .Net languages which compile to the Common Language Infrastructure. Thus, we must only add to the TCB the bytecode verifier, which has a smaller footprint and is simpler to proof check.

Following this lead, type safety protection mechanisms are being moved to low level languages in order to reduce the complexity of the verification of type safety preservation on the generated code. Specifically, Typed Assembly Language (TAL) [20] consists of a regular instruction set augmented with a memory allocation instruction and with type annotations. High-level languages are compiled into a series of typed intermediate languages and finally into TAL. The TALC compiler generates code for the Intel 32-bit family of assembly languages. This assembly code is annotated with type information that can be verified by the type checker before the code is assembled by the MASM assembler.

The two most significant contributions of TAL are:

- (**i**) It is largely source language independent and
- (**ii**) The transformations necessary after verification are minimal. In particular, TAL allows code generation optimizations before verification.

6. Language -Based Access Control

Techniques such as *access control* (static or dynamic) and *capabilities* are used to implement discretionary access control policies in programming languages.

a. *Static access control*. Via procedural encapsulation, a resource is a private field of an object, or a free variable of a function closure, either of which is given to the principal who cannot directly access it; the code of the function will perform the required access checks before performing the operation. Also via *type abstraction*. Abstract types (and subsumption) can be used to limit the operations that can be invoked on an object. Object-oriented languages provide *access modifiers* such as *private*, *protected*, and *public*, and *default to package*, to restrict the visibility of attributes and classes.

b.*Dynamic access control*. There are several forms to enforce type safety during execution of (untrusted) code: interpretation of source (if available), compilation of source with run-time checks, bytecode interpretation, and instrumentation of pre-compiled machine code with necessary checks[30]. To reduce dynamic type checking overhead, static checking is used when possible; with no source, language systems such as Java's JVM does static type checking of bytecode during verification.

Java and Microsoft's .NET framework provide a very similar security architecture for *code-level* access control to control the access code has to protected resources and allowed operations on them. This system is dynamic in that access restrictions are checked at run-time via the so-called *stack inspection*. This security architecture specifies functionality to define permission lists for code, configure the security policy, allows code to request permissions required in order to run, grants permissions during loading based on the permissions allowed by the code. Some resources such as the file system or network sockets are also protected with permissions.

To use this architecture, each method is associated with a principal, and the permissions granted are those of the least privileged principal appearing in the method call stack leading to the current operation. Specifically, the programmer adds "do privileged" and "check privilege" commands to the code. A "do privileged" command adds a flag to the caller's stack frame, which is eliminated when the frame returns. When a privilege is checked, the *stack inspection* algorithm searches the frames on the caller's stack in sequence from the newest to the oldest. If the search encounters an untrusted stack frame, which can never get a privilege flag, the search terminates, access is forbidden, and an exception is thrown. The search also terminates if a system stack frame with a privilege flag is encountered. In this case, access is allowed. For shared libraries needed by principals, privilege amplification mechanisms are provided. Note that the search and elimination of these flags implement a form of POLP, by making sure that code executes with the necessary privileges and no more.

c. *Capabilities*. An alternative to access control lists is capability-based security [16]. A *capability* is an **unforgeable** pointer to a controlled system resource. In the capability model, possession of a capability to some object is sufficient to use that object, and the reference distribution graph is at the heart of the security policy. Thus, a permission is encapsulated as a capability, and the complete set of system permissions is given by the system's reference graph.

To use a capability, a principal must have been first given that capability, as part of its initialization or as result of calling another capability. Once a capability has been given to a principal, the principal may then use the capability and, in some systems, may even pass the capability to other principals. This leads to a basic property of capabilities: any program which *has* a capability *must* have been permitted to use it. POLP is the overriding design principle in this model, an object can only perform the functions given by its present capabilities. System security is built around the concept of giving each principal only the authority necessary to perform the requested task; this is contrary to current software systems which grant principals excess authority for the sake of functionality, giving a successful attacker the same power the principal possesses. An attacker of a capability based system, will only get the limited functionality provided by the capability the attacker successfully penetrated. Thus, at worst, capability systems limit damage cause by an attacker.

Languages such as E [6], and Joe-E[13] implement capability-based security. E is a dynamically typed language, which defines and implements a pure object model of secure distributed persistent computation. Joe-E is Java based, as Java already provides a sound basis for capabilities through objects, as object references are non-forgeable; but the API must be redesigned for a full capability system, which in particular implies that no static functionality must be available and native methods are forbidden. Java libraries must be "tamed" to hide features incompatible with the capability model. Several E applications have been written and fully validated for security, including a web browser and a desktop [6].

It is worth mentioning that the capability model has been used in the design and implementation of several secured operating systems such as SPIN, KeyKOS, and J-Kernel[7].

7. Language -Based Information Flow

Systems may violate the *security* or the *integrity* of the system by releasing secret information or corrupting sensitive information. That is the reason why it is mandatory in many situations to control manipulations performed by a program in order to ensure they fulfill some integrity or security policy. Access control checks place restrictions on the release of information *but not its propagation*; they provide some protection but require the principal to which access is granted to be trusted without any further restrictions.

There is little assurance that current systems protect data confidentiality and integrity. Although practical techniques for enforcing these properties such as firewalls and encryption are useful for protecting confidential information, they fail to provide end-to-end security. A firewall protects confidentiality by preventing communication with the outside but permits some communication in both directions, while confidentiality leaks lie outside the scope of the firewall mechanism. Similarly, encryption can be used to secure an information channel between two communicating endpoints. However, encryption provides no assurance that once data is decrypted, the receiver respects the confidentiality of the transmitted data.

In general, covert channels [21] pose the greatest challenge in preventing information leaks. Their existence has led to a more general approach of information flow security: *Information flow analysis.* It consists in statically analyzing the source code of a program *before* its execution, in order to ensure that all operations performed respect the data flow security policy of the system. The idea is to try to directly control the whole flow of information, rather than the accesses of principals to objects. By placing information flow rules, it is possible to

control direct and indirect leakages, as in this perspective, they both become "unwanted information flows".

This analysis can be performed by enriching a **static** type-safe system whereby the types of program variables and expressions are augmented using a lattice of labels to specify policies on the use of the data. This static analysis allow us to reason about data flow policies with no run-time costs. The label order, along with a *program counter (pc) label* that tracks pc dependencies, are used to detect explicit or implicit information flow between expressions with different labels. As example, suppose `h` is an `int` variable with a high label and `l` with a low label; the code below is flagged illegal during analysis, as the pc label associated with it is high, while there is an assignment to the low label variable `l` in the `then` clause.

```
h = h mod 2;
l = 0;
if h == 1 then l = 1
else skip
```

Note that this code has the exact same effect as the explicit and illegal flow given by `l = h mod 2.`

Sequential type safe languages with information flow analysis via labels are *Jif* [14, 21] based on a Java subset, and *FlowCaml*[25] based on a subset of Objective Caml.

8. Security Of Program Extensions

Program extensions via the downloading of components from the web, so-called mobile code, is now common practice. Thus, host programs must protect themselves from possibly security mis-behaving mobile components. Programming languages have developed three models to deal with these kinds of extensions. The first model, "*sandboxing*", used by Java for applets, provided a very restricted execution environment for the applet whereby any resources needed by the applet such as files and network connections are allowed only from the site where applet originated. This restricted environment evolved to one where *digitally signed applets* — the all or nothing model—were fully trusted besides those that were not digitally signed and remain in the sandbox. The last evolution of this model introduced permissions associated with downloaded code associated with owners, along with the dynamic "*stack inspection*", described earlier, yielding more flexibility to extensions while preserving the security of the entire program.

Downloaded code also brought the need to verify it for the minimal safety policies stated earlier. In the case of Java, its bytecode verifier checks for such safety policy—type safety and stack safety—while at run-time its security manager, using stack inspection, monitors execution of only permissible code. Note that this process places the burden to check for the security checks of mobile code on the consumer.

Certifying compilers [18], were developed to place the burden of security abiding of extensions on the provider of the code. A *certifying compiler* is a compiler that, when given source code satisfying a particular security policy, not only produces object code but also produces a *certificate*—machine-checkable proof of evidence— that the object code respects the policy. Mobile code consumers, on the other hand, will only need to verify the proof. Certifying compilers are an effective tool for policy enforcement as they do not enlarge the TCB. Also certifying compilers gave rise to the specification and verification of security policies beyond the minimal one.

9. Language - Based Denial of Service

Neither type safe systems nor capability systems guarantee termination, but domain specific languages exist that employ computational restrictions to do so. The Packet Language for Active Networks (PLAN) [10] disallows programming loops of non-fixed length and the use of unrestricted recursion—the programmer must hard-code all looping and recursive bounds. Because this restriction ensures program termination, PLAN can be used for programming active networks, but not for DoS attacks because termination ensures that CPU and memory won't be consumed out of system bounds.

10. Security In Weakly-Typed Languages

Weakly type languages such as C/C++ do not meet the minimum criteria for system security, witness the different types of attacks possible with such languages (Buffer overflow, string format errors, integer overflow, etc). But C is the de facto standard for systems programming due to its low-level data representation, programmer's control of memory and efficiency. Besides the development of static and dynamic analysis tools for C to uncover vulnerabilities, several languages [15] have been designed for C with the right trade-off between safety and functionality, while minimizing or in some cases totally avoiding the effort of porting C code. The common features of these languages are the introduction of fat pointers (which include size) and a constraint form of memory management to track memory usage and deallocation. These languages also rely on static as well as dynamic checks.

SafeC [2] and *CCured* [19], focus on compatibility, with the objective to make legacy C code safe while avoiding a complete overwrite. *Cyclone* [12, 4] and *Vault* [8], are safe alternatives to C with novel constructs and idioms designed for systems programming, although legacy C code rewrite is unavoidable. Up to 15% rewrite in Cyclone examples [31].

SafeC changes the representation of pointers to an extended structure that contains safety information such as memory bounds. The compiler then transforms conventional C programs by converting pointer representations and inserting necessary run-time checks. It requires few modifications to C source code and is a good tool for legacy code safety. However it provides limited safety guarantees at a high price (130% to 540%) due to exhaustive run-time checks.

CCured, a type-safe implementation of C, statically attempts to verify that source code is free from memory errors, and introduces run-time checks where static analysis does not guarantee safety.

Cyclone, is a type safe variant of C, with fat pointers and region-based memory management [11]. It restricts it to preserve type safety: unsafe casts are disallowed, restricts pointer arithmetic, pointers are checked for null, unions are tagged. A fat pointer access is guarded by a dynamic check. Memory is divided into regions and programmer's allocation come from a region. This mechanism is based on the observation that in many situations we require a group of objects be deallocated at the same time rather than deallocating them individually at different times. To avoid dereferencing dangling pointers, Cyclone tracks the set of live regions at each program point, and augments pointer type with its region name. If a pointer's region is alive, dereferencing is allowed.

Microsoft's Vault, claims to be a safe version of C, like Cyclone, but is more like C#, with a module system and a novel feature for specifying and checking program resource use. Programmers are able to control data layout and lifetime of program's resources while being provided with safety guarantees.

Other authors [5] propose a hardware bounded pointer solution as the above languages all suffer from one or more deficiencies that may prevent wide adoption, such as: unacceptably high runtime overheads, incomplete detection of memory

bound violations, incompatible pointer representations (due to memory layout changes), or requiring non-trivial changes to existing C source. Hardware pointers facilitate software enforcement of bounds on memory by having the application use the new *setbound* instruction for communication of valid pointer bounds, while it checks and propagates bounds information as the pointer is manipulated.

11. Classroom Experience

This material has been used in our home institution in a three hour security topic within the Programming Languages course. It has provided our students with a sound foundation for the evaluation of security features in programming languages. Besides the main topics discussed herein, we explore the Java security architecture and take a closer look into the security capability model in the E language.

12. Conclusion

Type safe languages, and capability-based systems provide a very sound foundation for the building of secure systems. With the threat of attacks at the application level, the use of such languages rule out many attacks which are prevalent with the use of non-type-safe languages.

We have also seen how modern type safe language systems not only embody the minimal security policies, but can be enriched to provide support for other security concerns, such as information flow. Furthermore, they provide a sound foundation on which to model concepts that once were part-and-parcel of operating systems such as control-access lists, and runtime security checks. (Type safe languages such as Java are being used to model operating system's processes, and even protection domains at a finer level than operating system processes [7].) We must also add that the information flow languages Jif and FlowCaml, mentioned earlier, need further development as they lack features required by current applications, one of them being concurrency among others.

On the other hand, capability systems model and implement security via unforgeable references to objects which then allows us to provide to principals only the functionality needed, a strict implementation of POLP. This yields a minimization of functionality available to successful attackers.

In closing, type safety as well as capability programming are very effective in allowing system designers specify, implement and rigorously enforce security policies, while doing it at a much finer granularity than what is possible via operating system techniques.

13. Acknowledgments

We thanks our students and the NSF/CAE program for their support with this research. We also thank Vassil Rousev and Golden Richard for suggestions on this paper.

14. References

[1] R. Anderson. Security Engineering. 2nd. ed. Wiley. '08.

[2] T. Austin, S. Breach, and G Sohi. Efficient detection of all pointer and array access errors. Proc. '94 SIGPLAN Conference on Programming Language Design. 1994.

[3] M. Bishop. Introduction to Computer Security. Addison Wesley. 2005.

[4] http://www.research.att.com/projects/cyclone

[5] J. Devietti, C Blundell, M. Martin, S. Zdancewic. HardBound: Arch. Support for Spatial Safety in the C Prog. Language. Proc. of ASPLOS. March, 2008.

[6] http://www.erights.org/

[7] T. von Eicken, C. Chang, G. Czajkowski, D. Hu, D. Spoonhover. J-Kernel: A Capability-based Operating System for Java. LNCS 2000. Springer Verlag.

[8] M.Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative progr. In Proc. of the SIGPLAN PID Conf. Berlin, Germany, June 2002.

[9] C. Hawblitzel and T. von Eicken. A Case for Language-Based Protection, . Dep of Computer Science, Cornell University, Technical Report TR98-1670, March 1998.

[10] M.W. Hicks and A.D. Keromytis, "A Secure PLAN," Proc. 1st Int'l Working Conf. Active Networks, LNCS 1653, Springer-Verlag, 1999

[11] M. Hicks, G. Morrisett, D Grossman, T Jim. Safe And Flexible Memory Management in Cyclone. CS-TR-4514. Dept of Comp Science . Univ. of Maryland, 2003.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In USENIX Annual Technical Conference, 2002.

[13] http://code.google.com/p/joe-e/

[14] http://www.cs.cornell.edu/jif/

[15] P. Li. Safe Systems Programming Languages. Department of Computer and Information Science.Univ. Of Pennsylvania. 10/6/2004.

[16] M. Miller,B. Tulloc, J. Shapiro. The Structure of Authority: Why Security Is not a Separable Concern. Second Int. Conference Mozart/OZ . LNAI 3389. 2005

[17] J. P. Myers. Information Assurance and Security: Even mortals can teach it. BoF session. SIGCSE 2008.

[18] G. Necula. Proof-carrying code. In Proc. 24th ACM POPL, pages 106-119, January 1997.

[19] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrotting of legacy code. Proc. 29th ACM Symp. on P Portland, OR, January 2002.

[20] B. C. Pierce, editor. Advance topics in types and programming languages. MIT Press, 2005.

[21] A. Sabelfeld, A. Myers. Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communication, Vol 21, No 1,Jan 2003.

[22] M.. Scott. Programming Languages Pragmatics. Morgan Kaufmann. 2nd ed. 2006.

[23] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 9(63), Sept. 1975.

[24] R. Sebesta. Concepts of Programming Languages. Addison Wesley. 8th. ed. 2007

[25] V. Simonet. Flow Caml in a Nutshell. Proceedings of the First APPSEM-II Workshop. 2003.

[26] S. Smith. The craft of System security. Addison Wesley. 2008

[27] A. Tucker, R. Noonan. Programming Languages. 2nd ed. McGraw Hill. 2007.

[28] J. Viega, G.McGraw. Building Secure Software. Addison-Wesley. 2002.

[29] J. Vitek, B.Bokowski. Confined type in Java. Softw. Pract. and Exp. v.2.1, 2000. R.

[30] Wahbe, S. L, T. Anderson, and S. Graham, "Efficient softwarebased fault isolation," in *Proc. ACM Symp. on Operating System Principles*, Dec. 1993, pp. 203–216.

[31] D.Grossman et al. Cyclone: A type-safe dialect of C. C/C++ Users Journal. January, 2005.