Robustly Secure Computer Systems: A new security paradigm of system discontinuity

Jon A. Solworth*
solworth@rites.uic.edu
University of Illinois at Chicago

ABSTRACT

For over 30 years, system software has been bound by compatibility with legacy applications. The system software base, whether proprietary or open source, is dominated by the programming language C and the POSIX operating system specification. Even when commercial operating systems stray from this model, they don't go very far.

Unfortunately, the POSIX/C base was constructed in a more benign environment then today and before many security issues were widely understood. Rather than fix these issues, compatibility has been deemed more important than security, and so this base has been kept with its flaws intact. As a result, programmers routinely create software with security holes—even in the most security critical software—and today's systems are easily attacked.

We propose a new paradigm of *system discontinuity* which emphasizes security over compatibility by removing those constructs in our system software which lead to security holes in applications. Of course, removing parts of the interface will break applications, and hence the discontinuity. To deal with this situation, we advocate the use of virtual machines to enable multiple operating systems to run concurrently. Thus high security OSs can be used for the most security sensitive applications. Compatibility is maintained for less security sensitive applications using legacy operating systems. Over time, legacy applications can migrate to a more secure OS, thus raising the security of all applications.

1. INTRODUCTION

All of computer security rests on a foundation of correctness. If critical components of a computer system—such as authentication and authorization (access controls)—are not correctly implemented there is very little assurance that the system will work securely. The *Trusted Computing Base (TCB)* captures the notion of software components in which correctness is critical to system security.

The Orange Book [26] requires minimizing the vulnerability of the TCB in systems with high security ratings. At the highest level, A1, techniques are used both to reduce the size of the TCB and to minimize its flaws. For example, security kernels are mandated to reduce that part of the operating system kernel in which errors could result in critical vulnerabilities. In addition, automatic verification techniques must be used to validate the TCB, thus going to substantial lengths to reduce errors[1]. Other TCB requirements reduce the susceptibility to attack, for example by logging of software changes.

The Orange Book concentrates on the issue of confidentiality. Strict confidentiality is very well suited to a TCB approach: To maintain confidentiality, processes are (1) simply prevented from reading information for which there is no "need to know" and (2) once having read that information are prevented from writing to certain objects [3]. The issue of covert channels [22, 12, 11], although a thorny problem, can in principle be addressed within the TCB.

The issue of integrity is more subtle. A security kernel cannot enforce integrity. It can prohibit processes from writing files unnecessarily and it can ensure information flow integrity [4] is maintained. But it cannot ensure that processes write the *correct values*[2] into files to which they have write privileges—and hence cannot ensure that integrity is maintained. And thus integrity, unlike confidentiality, inherently depends on the applications. In fact, the Orange Book says: *"It must be understood that the completion of a formal product evaluation does not constitute certification or accreditation for the system to be used in any specific application environment."* But the applications are the purpose for the system.

For many systems, integrity is more important than confidentiality. Today, it is often the system integrity which is

*This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

[1]In theory, validation should result in removing, not just reducing, vulnerabilities. But there are inherent limits to validation which prevents this from occurring.

[2]The correct values depend on the computations of applications and are inherently unknown by the operating system.

attacked, for example, converting a system into a bot or appropriating a bank account. Moreover, integrity attacks are often the first stage of large-scale attacks such as Distributed Denial of Service and SPAM. Oddly enough, current popular Operating System (OS) authorization models do relatively little to protect system integrity [23].

Hence, the correctness of integrity sensitive applications is essential. But correctness assurance of a non-trivial set of applications is a formidable problem. To be able to do so would mean that there is a formal specification for what those applications must do. (Otherwise, application correctness would not even be defined.) But the formal specification of an entire set of applications would be complex, and *its* correctness difficult to determine. Therefore, it is only possible to deal with at most *some* correctness issues. Like the validation of a security kernel, a way must be found to make systems more correct in ways that significantly improve their security, but with the knowledge that failures will still occur [23].

Fortunately, not every failure is equally damaging. We have long known that damage in the TCB can be fatal to system security, like an injury to vital organs can be fatal to humans. But injury outside the vital organs can also be fatal. It is thus essential to remove vulnerabilities in security-critical applications.

Integrity attacks against sensitive applications are therefore a deep concern. The most damaging of these include code injection attacks, which dynamically substitute untrusted code for trusted code, for example by buffer overflow. Other integrity attacks exploit memory corruption errors, race conditions, Time-Of-Check To Time-Of-Use (TOCTTOU) inconsistencies, and error prone constructs. These bugs are (primarily) in applications, but the *pitfalls* which enable these bugs are in systems. Many of these pitfalls can be avoided by a careful system design. But since compatibility has been paramount, these pitfalls are almost never removed from systems. Instead, it is up to application programmers to avoid them.

The practice of expecting programmers to work around these pitfalls has failed: *"experience has shown that only a hand full of programmers have the right mindset to write secure code"* [47]. These security holes regularly show up in security critical code and in major products from the top (and richest) companies in high tech. Paradoxically, despite being difficult for defenders to avoid, attackers regularly exploit them[3]. Attacks are increasingly being discovered on the same day that the code containing a vulnerability becomes available—zero day exploits.

It is long past time when the IT community should have discontinued its error prone practices which can be exploited to cause extreme damage. It is amazing that programs are still written in type unsafe languages such as C/C++ despite our extensive experience that it is dangerous to do so. It is perhaps a sign of a lack of professionalism in our discipline, that we do not deal with our failures and take appropriate corrective action. In other fields, for example, aerospace engineering, each accident is examined and the lessons learned; these lessons are incorporated into future designs and advisories are created to mitigate the problem with current aircraft. Of course, aircraft accidents often entail loss of life, and so such care is essential. But do accountants, whose miscalculations are not normally assumed to be life threatening, ignore the impact of their errors?

We believe that the number of extremely dangerous bugs in fielded systems can be dramatically reduced—by at least a decimal order of magnitude. It is only possible to do this by carefully selecting the types of bugs we seek to eradicate.

Some of the techniques which prevent such problem-prone constructs do not necessarily reduce the number of bugs in the system. For example, buffer overflows can be eliminated by having the program abort rather than buffer overflow: but this is *still* a bug and can result in a security vulnerability, denial of service. However, it does prevent a hacker from gaining control of the system. Moreover, this loss of functionality is much more likely to be visible to the system's owner and thus is easier to identify and correct. Hence, such techniques reduce, on average, the damage that bugs do.

We draw inspiration from reliable systems. Perhaps the most widely used model is *fail fast* [14], in which errors are detected and then immediately cause a system restart before system state can be corrupted. An example of this technique is an OS panic-induced reboot, in which the OS code detects an anomaly and then reboots itself. Fail fast works because it reduces the complexity of systems by restarting from a valid state rather than attempting to repair a faulty state. It avoids the much higher complexity of failed systems (which have many more states than good systems) and relies instead on a smaller recovery mechanism which is, in any event, needed.

Of course, in a safety critical function, such as the control of an airplane, denial of service may have very serious consequences. Even so, the integrity threat is usually worse. If an attacker exploits a buffer overflow, it may trigger a denial of service (by triggering an abort); it may also perform numerous other attacks. Detecting the errors which can create a denial of service is expensive, and requires extensive testing. For this reason, it is impractical to expect that denial-of-service testing will be universally performed. Furthermore there is no practical way for a system owner to determine that such testing has been done (or done effectively) on software obtained from other parties. At any rate, experience has shown that it is undesirable to assume that the quality of application code will always be high.

Authorization and authentication can play a significant role in preventing the exploitation of application vulnerabilities. Unfortunately, given current mainline OS authorization systems, an attacker can gain control of the system in at most two stages. First a user–level process is broken into and second, a root–level process is broken into from the inside. This two stage attack is possible because the OS does not enforce least privilege [30], particularly in regard to the privileges

[3]For example, Slashdot had a story `http://it.slashdot.org/article.pl?sid=04/12/15/2113202` about a course taught by Daniel J. Bernstein at UIC on Security Holes in which 20 students found some 44 exploits in publicly distributed software. (One student found 10 exploits).

of an executable. Reducing a process's privileges increases the number of steps on average, and hence the number of exploits, needed to gain system privileges.

The new paradigm proposed here is called *system discontinuity*; its purpose is to improve systems and especially the applications built on top of them. This discontinuity is caused by changing those interfaces—both of programming languages and operating systems—which have proven to engender the greatest number of security holes. In particular, we focus on those interfaces that lead to a non-trivial number of integrity or confidentiality failures. Changing existing interfaces will break applications which depend on them, and hence the term system discontinuity. But once these applications are ported, they will be more secure and less expensive to maintain since they will have fewer of the most dangerous flaws. These techniques will also make our systems more reliable.

This strategy is a departure from decades of emphasizing compatibility over reliability and security, but we believe it is necessary as our current computing base is failing (see Section 2). For this strategy to be successful, it is essential to minimize the dislocations from system discontinuity. In particular, we need to avoid a fundamental chicken-and-egg problem in new systems design, which we dub the *application trap*: No one uses new systems because there are no applications for them and there are no applications for them because no one uses the system. If this (vicious) cycle can be broken, then a self sustaining economy can be created for a new system.

We believe that the escape from the application trap is made possible by Virtual Machines (VMs). VMs can play a pivotal role, by enabling many different versions of operating systems to be concurrently executed on a computer. The most security sensitive applications will, of course, be ported first to the most secure OS. New applications will bypass older OSs and be built on the most secure OS available. Application compatibility with older OSs is no longer necessary in the age of VMs.

We have been pursuing this approach, in a small way, by replacing the authorization model (access controls) in Linux with an alternative one with far stronger semantics [28, 29]. The change consists of about a half dozen system calls, such as `chown` and `chmod`. While these system calls are semantically important, they are not very heavily used. In one semester, two students ported 3 programs (`xpdf`, `bash`, and `thunderbird`) to use the new system calls; the last of these programs contained over a million lines of code. The students had no prior experience with these applications. We conclude from this exercise that porting such applications is quite manageable and may further benefit from the development of tools.

But incremental change will not be sufficient to reach the far more secure systems envisioned here. Many problems have accumulated over 30 years of maintaining backward compatibility, and substantial change is now necessary. We advocate one big change, called here the "great leap" (see Section 3), to address this long neglect. After that, incremental changes can be made, and applications ported with the help of tools.

Pursuing this strategy will be expensive. The size of the problem and some of the cost measures are examined in the next section. Despite the costs, we believe that system discontinuity is necessary, as we can see no other way to achieve sufficiently secure systems given the current threat environment. To paraphrase Sherlock Holmes when all the other options have been eliminated, the remaining option, no mater how unlikely, must be embraced[4].

2. ECONOMICS

Economics plays an important role since what we propose—the wholesale replacement of the current infrastructure with a much more reliable one—will be a very expensive undertaking. It will also take significant time, so it is important to begin work on it soon.

The computing base is extremely vulnerable. For example, the annual occurrences of vulnerabilities, as reported to CERT[5], are shown in Table 1. The number of reported vulnerabilities has grown from 171 in 1995 to 8,064 in 2006. This is an alarming trend. We believe that the results indicate that (existing) bugs are being discovered at an ever increasing rate. But software also keeps changing, and there is anecdotal evidence that the change itself is a primary source of security holes. And hence, there are an increasing number of zero day exploits, even in security "fixes". The computing base is riddled with security holes.

There is no magic bullet, now or visible on the horizon, to automatically remove these holes. These attacks elude authentication and authorization by directly attacking the correctness of applications. And yet authentication and authorization cannot be discounted as defensive mechanisms, as they remove avenues from attack and increase the work required by an attacker to breach a system. Hence, new system designs must remove the pitfalls which lead to security holes, improve the quality of authentication, and use authorization to increase the number of exploits an attacker needs to abridge system security. We believe that systems built on such principles could have a profound effect on the entire software base.

It is necessary to replace the software because of the steadily increasing level of attacks aimed at our computing base. These attacks are, in large part, financially motivated. The attackers are professional, with some gangs estimated to be pulling in over $1 million/day. They are using their illicit profits to hire security experts to craft automated attacks, to keep ahead of the defenders, and to increase their income.

Far from being centralized, the high tech attack-software industry is highly distributed. It consists of many different specialties, with boutique organizations catering to various segments such as phishing software, botnet renters, and sellers of keystroke logs and credit card numbers, etc. The key metric is the number of bots since this measures compromised desktops; estimates range from 1.5 million to 150 mil-

[4]Sherlock Holmes "When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth."

[5]`http://www.cert.org/stats/cert_stats.html`

1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006
171	345	311	262	417	1,090	2,437	4,129	3,784	3,780	5,990	8,064

Table 1: Vulnerability reports to CERT from 1995-2006

lion computers—somewhere between .25% and 25% of the installed base[6,7]. The attacker base is decentralized (and thus hard to thwart), has an economically sustainable model, is organized along the lines of traditional high tech industries (and hence is very flexible), and is located overseas (beyond the reach of law enforcement of the victims). To date, our response has been totally inadequate.

The attacks are aimed primarily at the weakest link, the desktop. It may give comfort that the high value servers are not the direct target. But it is not possible to protect the servers when the desktops are compromised, as the servers are accessed through the desktops. While a given bank may be able to escape blame—and financial responsibility—for their customer accounts being raided, this in no way lessens the damage inflicted. And ultimately, the whole banking industry is at risk. These attacks show a dramatic increase in level starting in 2002-3 as the number of desktops with high speed Internet connections increased dramatically.

The central question is: How big is the problem? We don't know. The costs includes the losses due to theft, espionage, and extortion; remediation of compromised systems; and the loss of utility from systems which are not fielded because it is too difficult to secure them.

One Secret Service agent estimated that credit card fraud alone was running at $100 Billion/year[8]. Other financial frauds are against bank accounts and brokerage accounts. Espionage is aimed at national governments and at industry, particularly for intellectual property. Extortion is the preferred method for those who commit Distributed Denial of Service—there were recently 3 individuals convicted in Russia for attacks on Great Britain gambling sites resulting in millions of dollars in losses[9]. The FBI has estimated US losses to cybercrime at $67 Billion/year, it is likely that the true cost is even higher as many organizations have incentive to underreport these problems[10]. These attackers are appropriating our bank accounts and investments, our infrastructure is increasingly under their control, and our intellectual property and privacy lost. The stakes are very large and the threat is growing.

What would be the cost to fix the problem? The cost of a new operating system has been estimated at over $1 Billion; the cost of porting applications to it will likely be orders of magnitude more. But the cost of getting started is far less. It is estimated that it takes five years to build an operating system and an additional five years to make it stable. Such an effort would require a small team, costing $10 million for the first stage and $10 million for the second stage per operating system[11]. Clearly there would be more first stage operating system designs than second stage fundings, since only the best ideas would move forward. After this 10 year effort, the pump would be primed, and the larger community and economy for these operating systems and their applications could develop organically. Such an approach is only feasible using VMs, which would enable these new systems to be fielded along side of existing ones and enable a gradual transition to the more secure systems.

We note that the above estimates of time, and especially cost, assume a traditional model of OS development. It is highly desirable to lower the costs and decrease the development time, attaining earlier critical mass in which a self-sustaining economy is established for the software. If development could be specialized for the need of building security enhanced interfaces, significant speed (and cost savings) might be achieved during the first phase, so that the right ideas could be found much more quickly.

We are currently exploring techniques by which this might be possible. Our goal is to once again leverage the VM infrastructure, this time by sharing I/O devices from other Operating Systems running within the same hypervisor. We will describe this technique in a forthcoming paper.

At any rate, we believe that the current threat environment and the relatively low cost of investigating alternatives justifies a substantial effort.

3. THE GREAT LEAP

We believe that a new generation of security-aware systems must be developed. They will, of necessity, be incompatible with the existing systems which are a major cause of insecurity. A reasonable goal is that these new systems be at least an order of magnitude more secure than existing systems. Thereafter, more incremental improvements can be made. The focus in this section, is on the "great leap" to more secure systems, which will provide the largest improvements in security and the greatest discontinuity in the Application Programming Interface (API).

What's not being proposed. Before describing what we believe should be done, we want to be explicit about what we are not proposing to do. Computer security cannot be provided totally at the OS/programming language level. For example, the web is today a larger source of reported vulnerabilities [7]. Web issues include how to deal with pluggable architectures which enable code to be included from disparate sources and segregation between different sites (e.g. XSS). These are the same issues, in different guises, as those

[6] `http://www.avertlabs.com/research/blog/?p=185`

[7] Such a wide range of estimates, 100:1, makes it impossible to appropriately allocate resources to defend against this problem.

[8] CCS'06 panel on phishing and botnets.

[9] `http://www.informationweek.com/news/showArticle.jhtml?articleID=193104471&subSection=Breaking+News`

[10] `http://news.com.com/Computer+crime+costs+67+billion,+FBI+says/2100-7349_3-6028946.html`

[11] These estimates were made by Tom Anderson of University of Washington.

encountered in an OS. It may be possible to better support the needs of the web by relying on improved OS facilities, resulting in a more integrated security framework. Examples of this interplay between pluggable platforms and OSs include the use of VMs [8, 9, 45], SubOS [16], and Sandboxing [44, 13, 19] to significantly improve the security of browsers. In addition, because security is a matter of the weakest link, an integrated framework is important as it prevents gaps from occurring in the protections. In any event, the lessons learned when applying system discontinuity to an OS can be also applied to the web.

Other vulnerabilities due to insufficiently checked inputs, monolithic programs, etc. will continue to plague us. Additional vulnerabilities arise from economic externalities, in which the entity which causes a security problem does not bear the cost of the resultant security breaches and hence appropriate expenditures for security are not made [2]. Economic externalities argue that control of system security should return to system owners. The owners can and should be provided with simple tradeoffs by which they can secure their systems.

Semantics for new systems. The big question is: What should the semantics be for these new systems? Clearly, the errors of the last 30 years should be avoided. Hence, it is crucial to understand the source of past security holes. In particular, for each security hole it is desirable to understand the:

error which is the (initially latent) bug,

environment which is the semantics enabling the bug,

fault which is how the bug is triggered, and

failure which is the types of security breaches that could result.

Unfortunately, the error-environment-fault-failure information is often not available in the security vulnerability databases. It is important that such information be collected in the future, and to the extent possible, derived by analysis from past reports. Because of the lack of sufficient data, the issues focused on below are, to a certain degree, guesswork and will need refinement. Our goal is to deal with the environmental issues which lead to integrity or confidentiality critical bugs.

This is a much broader approach than taken by the traditional secure operating systems community, which has focused on writing operating systems which (1) have lower rates of security failures and (2) stronger authorization models, for example [38, 10]. As far as we know, we are the first to look at system design through this more holistic prism. The closest to our OS approach is Microsoft's Singularity project which is focusing on building a reliable and secure OS at the expense of compatibility [15], although it appears to be inwardly centered around a microkernel approach.

The new systems interfaces should ensure that security vulnerabilities either cannot occur or are far less likely to occur. Of course, we prefer the former to the latter, but we believe both will be necessary to achieve the very substantial improvements sought. This is analagous to a highway system; a highway is designed to be safe at high speed, not just for the best drivers but also for poorer drivers, distracted drivers, poor weather conditions, etc [39]. Currently, our systems are unsafe at any speed[12].

To understand the types of changes which might be necessary in the great leap, we describe an initial set of fundamental interface changes summarized in Table 2. The issues will be explored in more detail in Sections 3.1–3.8. The first five items in the table are to prevent security holes from unintentionally being inserted into programs[13]. These changes are at the programming language and operating system level. The remaining items are designed to prevent security holes from being exploited, as well as to provide tighter controls on who can do what.

The 8 items in the table can be grouped according to their impact on application bugs.

improve testing effectiveness by decreasing the variability of how programs execute. For example, a buffer overflow or a race condition can have very surprising behavior and are difficult to find by testing. Items under this category include 1, 2, 3, and 4.

reduced complexity by removing constructs whose security implications are difficult to understand. In particular, those constructs that require determining whether there exists a sequence of operations that can result in an exploit are routinely sources of serious security holes. Items 1, 2, 3, 4, and 5 reduce complexity which affects the security properties of the system.

security by default by making it easier to build applications which provide isolation, authentication, and authorization than to build applications without these properties. Items 6, 7, and 8 address this issue.

fault isolation mechanisms keep the attackers at bay. They are items 6, 7 and 8. They are necessary even when there are no security holes. When there are security holes, they serve to keep attackers away from vulnerabilities, and thus render it much harder to exploit security holes.

Structural enforcement. An important advantage of this approach is that these restrictions can be enforced by the structure of a system. For example, type safety can be ensured by requiring the source code for executables be limited to specified programming languages. This is easy to do for open source, and can also be done for closed source by relying on guarantees from suitable parties. The operating system based interfaces are even simpler to ensure, since it is impossible for the application to use systems calls which do not exist.

[12]"Unsafe at any speed" was the title of the 1965 book on the car industry by consumer advocate Ralph Nader.

[13]They may also prevent some types of intentional attack.

Item	Issue	Problem eliminated
1	**Strong type system**	code injection attacks/memory corruption
2	**Sequential semantics**	race conditions, synchronization errors
3	**Atomicity**	TOCTTOU errors.
4	**Well defined semantics**	implementation dependent behavior.
5	**Error prone constructs**	the ability of untrusted processes to modify the environment of more trusted processes.
6	**Network security by default**	confidentiality, integrity, and authentication attacks against networks.
7	**Authorization**	the ability of newly installed programs to pose problems to existing programs, propagation of attack.
8	**Strong authentication**	anonymous attacks.

Table 2: Issues which need addressing

The approach advocated is easy for an organization to verify. In contrast, mechanisms such as model checkers for verifying that existing codes do not have latent security holes are much more difficult to use, since these mechanisms are subject to false positives (or even worse, false negatives) [6, 35, 25]. The false positives require manual checking which is impractical for almost all organizations, and false negatives mean that there are remaining holes. Verification remains an important mechanism for the creators of software, but is less appropriate for the users of software.

Minimizing discontinuity. Discontinuity is inherent in our scheme, but to the degree possible, it is desirable to minimize it. New code will obviously use the new APIs. But how can existing applications be ported to newer systems?

We obviously cannot switch the programming language employed overnight. However, applications can gradually be ported to new programming languages—by introducing new programming languages into the application while retaining existing ones.

However, eliminating APIs from the operating system will break existing applications. An incremental scheme will be useful here as well. As mentioned previously, VMs allow a gradual migration of applications, as the old, less secure operating system is run at the same time as the more secure one. System security will be improved to the extent that the most security-sensitive applications have migrated to the more secure OS. (This migration process can be repeated for more incremental changes after the great leap, so that multiple generations of secure OSs and their applications can run concurrently on the same hardware.)

This is a new paradigm since it goes against the last 30 years of OS design, in which compatibility has been paramount—both for UNIX and Windows—while security is treated as an add on. But compatibility above security is a broken model; rather than catering to the least common denominator, it is necessary to *raise* the level of our systems.

Of course, we will need to verify that operating system's and compiler's intended properties hold. Nevertheless, a combination of much stronger authorization, authentication, and elimination of the largest application vulnerabilities will make systems and their applications much safer.

Space and the early stage of this work preclude presenting an operating system design at this point. Instead, we examine next the items from Table 2 in more detail which, we believe, are central to solve for the great leap.

3.1 The type system

Languages which are not strongly typed, such as C/C++, have been the greatest source of system security holes and have enabled the most damaging of attacks. The most common exploit is a buffer overflow, which is a code injection attack. Another hole in the type system results from explicit deallocation of memory, resulting in bad pointers.

These types of bugs are difficult to find because they break the programming language abstraction. That is, the semantics of the programming language do not specify what will happen; instead, the underlying implementation must be examined to determine what is possible.

Type safe language, such as Java, C#, Cyclone, and bitC, enable the elimination of almost all type system errors [18, 36]. The few remaining sources of type errors are due to errors in the compiler or programming language's runtime support—these issues can best be addressed with validation approaches (similar to verification in A1 systems).

It is sometimes argued that the run-time overhead of type safe languages—which is estimated at between 50-100%—is too high for systems programming [37], because of power-limited machines such as servers and handhelds. Perhaps. But we are primarily concerned with the desktop, which is not typically running at capacity—far from it. And these are exactly the systems most easily attacked because of their large variety of software and lack of sophisticated administration.

We are not the first to make the argument for type safe languages. But type safety, while necessary, is not sufficient. Given current vulnerability reporting, it is difficult to know what percentage of bugs are due to type safety issues [7]. Although buffer overflow shows up in the top few entries, there are other issues which appear related, such as integer overflow/underflow and format string vulnerabilities. But even if one looks at the top 10 bug categories, they account for only 32% of open source and 42% of closed source reported bugs. Hence, dealing only with unsafe programming languages seem unlikely to make our systems ten times more

secure. It is necessary, therefore, to consider other parts of the system which contribute to the occurrence of security holes.

3.2 Sequential semantics

In its purest form, the process model in operating systems is a sequential model; its non-determinism (if any) arises only from interactions with the world outside the process. In such a model, what a web server does given a particular HTTP request depends only on what requests were previously processed. Requests may come from different clients and hence their (arrival and) execution order is determined outside the process. This pure process model leads to maximum repeatability of execution, enabling bugs to be effectively isolated and erroneous code to be identified and repaired.

On the other hand, explicit concurrency leads to problems of synchronization errors, deadlock, starvation, and race conditions. As in the case of type system errors, these problems pierce the high-level abstractions since race conditions can only be fully understood at the implementation level[14].

Concurrency bugs can be extraordinarily difficult to find by reading the code. They can also be extraordinarily difficult to debug, as the executions which trigger the bug may be based on an unusual confluence of events and hence occur rarely. They are called *heisenbugs* because they are non-repeatable [14]. The heisenbug effectively disappears once observed, and for that reason they are very difficult to debug. In addition, latent concurrency bugs may lurk for decades in code but only become exploitable with changes in hardware; for example, newer dual core processors mean that interleaving is increasingly fine grained, exposing bugs which were previously latent [46].

Race conditions can occur at the programming language, operating system, or application levels:

Programming languages may have semantics such as threads, which allow an enormous number of possible executions to arise, even when executing a program on a fixed input.

Operating systems have constructs such as threads, shared memory, and signal handlers which introduce concurrency into a process.

Applications may be implemented with multiple processes.

The removal of this explicit concurrency reduces the complexity of programs, thus making their semantics clearer.

Explicit concurrency within a process, whether implemented in the programming language or operating system is unnecessary. Implicit concurrency, on the other hand, comes from parallelizing sequential code [40]. Since the code is sequential, its semantics is sequential, and hence has none of the problems of explicit concurrency. Consider two sequential statements $s_1; s_2$, meaning that s_2 begins execution only after s_1 completes. However, s_1 and s_2 can be executed concurrently (written $s_1||s_2$) with the same result as a sequential execution if neither of s_1 or s_2 writes a location that the other statement accesses.

We do not rule out concurrent execution, only concurrent semantics; there are many models of concurrent execution with sequential semantics including in architecture (e.g., scoreboarding [42] and the Tomasulo algorithm [43]) and databases (in the form of transactions [14]). Without explicit concurrent semantics, systems would be significantly less complex and have fewer bugs.

Even in the case of applications composed of multiple processes, in which concurrency is inherent, the issues of concurrency are reduced through the careful selection of integrated communication and synchronization primitives (e.g., message passing or pipes).

3.3 Atomicity

A set of operations is *atomic* if they are either all executed (without other intervening operations) or none of them are executed. Atomicity can make the concurrent semantics of applications better behaved by preventing TOCTTOU errors. TOCTTOU errors have turned up in many different types of security sensitive software, including Kerberos, OpenSSL, Apache, and Samba [46].

To eliminate almost all forms of TOCTTOU errors, multiple system calls by a process could be made atomic, by providing general purpose transactions to processes. In addition to addressing TOCTTOU errors, transactions are a robust base which enable clean recovery from crashes and hence lead to more reliable systems. (One of the Orange Book requirements for higher rated systems is for security to be robust over crashes).

3.4 Well defined semantics

Programming languages and operating system semantics have typically been only partially specified; many details are implementation dependent. For example, in C/C++ the size of primitive types and the order of expression evaluation are implementation dependent. Java, on the other hand, specifies more of the semantics, including, the primitive types and expression evaluation order. However, Java's threads execute differently on different architectures (even without any external triggering events) and hence Java has gained the reputation of "Write once, test everywhere".

The issue of well defined semantics exists not only at the programming language level but also at the OS level. Many of the OS APIs are defined by POSIX [1] (for both UNIX and Windows). Some POSIX features are optional, others implementation defined, and others have defined variants. Furthermore, the error messages that can be returned by POSIX APIs are also implementation dependent and hence code must take into account all of the possibilities.

Such partially defined semantics mean that testing must be inadequate, severely limiting the ability of programmers to uncover incorrect semantic assumptions.

3.5 Error prone constructs

[14]The execution of a machine instruction is atomic in a computer system, the execution of a programming language statement is not.

Constructs which place the onus on the application programmer to reason about complex attack patterns are problematic. For example, the ability to remove a file under Unix depends only on whether one has write access to the directory containing it, but not whether the user owns the file being removed. This can result in substitution attacks, in which one file is substituted for another.

Indirection can also be a problem, as in soft links. A soft link is a directory entry which points indirectly to another file. Changing the link may result in a different file being read/modified, as an attacker may have sufficient permission to change the link but not the file itself.

To test designs for the presence of error prone constructs, model checking could be useful to determine whether there is an attack sequence using the constructs [46].

3.6 Network security by default

Network mechanisms were designed without any security, as networking was initially used to connect a limited number of homogeneous sites. Now, of course, networking is worldwide and extremely heterogeneous.

OS mechanisms are needed which, by default, use cryptographic protections and strong network authentication. (The current use of passwords over a network for performing network authentication allows password guessing attacks from anywhere in the world.) Examples of mechanisms which can be used for this purpose are distributed firewalls [17] and organization-based network authentication (e.g., [24]).

This will make it easier to implement network security in applications (since it would merely involve using the right OS primitives), and would make network security far more robust.

3.7 Authorization

Authorization limits an attacker's access, thus prevents exploiting vulnerabilities and restricts the actions of users. Role-Based Access Control (RBAC) have proven very effective at controlling access at the enterprise level [33, 32]. Strong and easy to use mechanisms, incorporating RBAC protections, are needed which effectively protect at the system level.

There are two central integrity issues for authorization. The first is to implement least privilege, so that when a process is compromised it yields as few permissions as possible to the attacker (and hence is the least harmful). This means limiting the privilege of a process based not only on the user who invokes it but also on the executable. Least privilege increases the average number of steps to attack a system, and thus increases the work load of the attacker. The second, is to prevent untrusted objects (e.g., email attachment) from getting close to critical applications [4, 29, 47].

3.8 Authentication

Authentication mechanisms are weak, thus providing too many opportunities for flaws in one application to spread into attacks on the system, and ultimately to obtain the root privileges. For example, a recent `ssh` attack used simple password guessing to break into many accounts[15].

Using mechanisms such as Kerberos [41] or public-key based authentication [31], it is not necessary to ever authenticate via a password transmitted over the network. An attack on this scheme would require an attacker to compromise a system which has the needed secret keys, a significantly more difficult task than a brute force password guessing attack.

4. HISTORY

Oddly enough, rather than security getting better over time, its gotten worse. Not because the attackers have gotten more skilled—although they have—but because systems have often been weakened with the introduction of new features. These features have been deemed desirable because performance has been placed over reliability. The weakening of system reliability was enabled by historical forces which made it possible for systems to evolve to their current state.

First, systems in the 1970s were relatively small and hardware was far less reliable than it is today. Techniques such as segmentation, error detecting and correcting codes, and processes were designed to limit the repercussions of hardware failures, by preventing the propagation of errors.

With increasing integrated circuit integration, hardware became steadily more reliable. Segmentation gave way to large, flat address spaces. Many techniques were added to operating systems including multi-threading in processes, shared memory, and an ever growing complexity. These techniques would not have been possible on earlier hardware, not only because of lack of resources but because they would have resulted in noticeably less reliable systems.

As memory became cheaper, very large processes were constructed, sometimes performing only loosely related computations. But these amalgamated processes require the union of the privileges of their components—and hence are more lucrative targets. Simultaneously, they also provide more attack points and hence are easier to compromise. We are beginning to see the reversal of this trend with privilege separation [27, 5].

The fundamental problems of security were well known in the 1970s [20]. What has changed is the diversity of applications and the accessibility of computer systems.

The attack profile started to change with the Internet Worm [34]. Hardware failure rates and attacks on software are different, but they share a common propagation model. As previously described, commonly a two step attack is sufficient to gain system-level privileges. Hence, security hole exploitation would be made more difficult if it is harder to propagate the attack towards an important target. As error propagation becomes more difficult, systems will be both more reliable and more secure.

We believe that it is therefore necessary to remove many of the "enhancements" to systems which have served to make them more vulnerable to attack. On the other hand, new

[15] `http://www.securityfocus.com/infocus/1876`

mechanisms are needed such as better network security, authorization, and authentication to control access to these systems. Most of all we need a virtuous cycle which will continue to weed out the pitfalls in the most effective ways and to enable us to get the greatest value from our software base.

5. WILL THIS REALLY ACCOMPLISH ANYTHING?

The class of bugs that will be severely curtailed by the system discontinuity paradigm are particularly nettlesome. It is difficult to test for their existence and/or difficult to debug them. They are a notorious source of security reports.

We would like to quantify the impact on system security, but it is difficult to do so for several reasons.

1. The data about security issues is sparse, in part due to commercial sensitivities. For example, the raw data is unavailable for the one recent extensive study that we have access to [7].

2. It is difficult to map current security vulnerability reports into environmental causes (the pitfall) and effects—whether the resultant failures are of confidentiality, integrity, or availability.

3. A substantial number of the bugs in the aggregated reports are either *unknown* or from a category *other* than those listed.

	unknown	other
open source software	10.1%	39.4%
closed source software	45.0%	12.4%

   In addition, information is not sufficiently specific. That is, it is not available by operating system, by distribution, etc. The information is also incomplete.

4. Even if all of the vulnerability types commonly exploited were removed from systems, these represent only the low lying fruit. Attackers would move onto the next level of more difficult to exploit vulnerabilities. Given the stakes involved, the need for continual improvement in systems seems unavoidable.

Hence, we don't know how much more secure these systems will be or even if the changes we propose address the main problems. But at the heart of this new paradigm is the application of engineering disciplines to the building of software. Thus, first and foremost, we believe it is necessary to start a virtuous cycle of improvement of the system software interface, based on experience (error reports), so that our systems empirically incorporate best practices. Attention to process is essential, but it is far from sufficient. It will be necessary to build better applications, to use authorization and authentication effectively, and to reflect the history of our discipline in best practices education and standards. Most of all, we believe in a process that eliminates those issues which have proved to be problematic while emphasizing simplicity of good design.

But will these vulnerabilities be replaced by a new class of yet more subtle bugs? No doubt, new exploits will be found. But the removal of large classes of bugs will significant improve our systems. Our goal is to increase the cost in both time and money to attackers who wish to compromise systems.

We will also need to carry on the following activities: Research is needed in the semantics and implementation of highly reliable operating systems—both how to construct them and most importantly, the design of their interfaces to support secure applications. Finally, techniques are needed to transition to new, more secure, systems. Innovative uses of VMs, program analysis and transformation techniques, etc., are needed to reduce the size of the discontinuities and smooth the transitions.

Most of all, we need to become very conservative in our fundamental platforms, whether they are operating systems, browsers, or scripting languages. There is a saying about pilots: "there are old pilots, and there are bold pilots, but there are no old bold pilots". It may be necessary from time to time to deal with tricky programming issues, but programmers should not seek them out. While we cannot force programmers to write good code, we can eliminate the system interfaces to write bad code. And best of all, we can do this with a relatively small and highly skilled part of the programming community.

It would be good, too, if the ideas here of leveraging more highly skilled parts of the community to affect the entire programming ecosystem could be extended to other groups besides system developers. Perhaps it would be possible to do so for application packagers, but that is an issue of future research.

6. CONCLUSION

We are proposing to reorient the traditional system development environment from maximizing compatibility to increasing system security and reliability. This is a methodology to be adapted from other disciplines, especially engineering ones, in which past errors are studied and their lessons incorporated into future systems. We call it *system discontinuity* since it involves breaking software in order to make it better.

This new paradigm of system discontinuity is enabled by the availability of Virtual Machines which allow multiple OSs to coexist on a given computer. Hence, application-poor but security-rich OSs can coexist with application-rich but security-poor OSs. This increases security to the extent that the security-rich OSs contain the most security critical applications. VMs have been used before to provide security using different domains (authorization environments) for the same system software [21], but we believe we are the first to advocate using VMs over different system software bases to improve security.

There will be considerable dislocations from a shift to system discontinuity, and considerable cost. However, attackers are doing extensive damage to our computing base, thereby degrading its value. The damage seems to be increasing in terms of the number of systems compromised. Moreover, the attackers are growing in wealth and sophistication; in short, they have a sustainable economic model. Defenders,

however, seem to be complacent about the current situation, attempting to apply patchwork solutions. But as is well known, security is a matter of the weakest link. Hence, a patchwork approach results in substantial number of vulnerabilities which can be exploited with disastrous effect.

The "weakest link" poses significant challenges for defenders. To deal with this challenge, we have identified a limited number of places—programming languages and operating system interfaces—where changes can have very broad effect across the whole spectrum of computer software. By building a better environment, many of the pitfalls which plague even experienced programmers can be eliminated, and our systems can fail much more gracefully. In addition, we need to provide new, highly usable mechanisms which dramatically simplify the building of highly secure applications, so that security is provided from the start.

Of course, it is not possible to fix all bugs and so we must be selective. We propose to fix what we believe are the most egregious of these problems, those that effect the confidentiality and integrity of our systems. (We really don't have a very good handle on the statistics here, more needs to be done to better understand them.) A shift away from problematic operating system APIs and programming languages will reduce the errors that unwary programmers fall into, thus making the vast majority of programmers far less prone to create system vulnerabilities.

Acknowledgements. The author thanks Manigandan Radhakrishnan, Satya Popuri, Ashwin Ganti, and the (anonymous) reviewers for their many suggestions which improved the quality of this paper. Special thanks goes to the New Security Paradigms Workshop, and its social compact, which results in a safe atmosphere for discussion and which stimulated many thoughts for future research.

7. REFERENCES

[1] IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language], 1996.

[2] Ross Anderson. Why information security it hard: An economic perspective. In *17th Annual Computer Security Applications Conference*, 2001.

[3] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, July 1976.

[4] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.

[5] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS*, pages 322–335. ACM, October 2006.

[7] Steven M. Christey. Vulnerability type distributing in CVE. `http://cwe.mitre.org/documents/vuln-trends.html`, October 2006. Mitre Corporation.

[8] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society, 2006.

[9] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.*, 2003.

[10] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005.

[11] Virgil Gligor. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, Ft. George G. Meade, Maryland, U.S.A., November 1993. Approved for public release: distribution unlimited.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[13] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. of the USENIX Security Symposium*, San Jose, Ca., 1996.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[15] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[16] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan Smith. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, 2002.

[17] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and Communications Security*, pages 190–199. ACM Press, 2000.

[18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *General Track: USENIX Annual Technical Conference*, pages 275–288, June 2002.

[19] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *SANE 2000*. NLUUG, 2000.

[20] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the multics security evaluation. *acsac*, 00:119, 2002.

[21] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM

security kernel for the VAX architecture. In *Proc. IEEE Symp. Security and Privacy*, pages 2–19, 1990.

[22] Butler Lampson. Protection. In *ACM Operating Systems Review*, volume 8, pages 18–24. ACM, 1974.

[23] Peter Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrel. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information System Security Conference*, pages 303–314, 1998.

[24] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, 1987.

[25] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst*, 27(3):477–526, 2005.

[26] Department of Defense. Trusted computer system evaluation criteria. Technical Report DOD 5200.28–STD, U. S. Department of Defense, 1985.

[27] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242. USENIX, August 2003.

[28] Manigandan Radhakrishnan and Jon A. Solworth. Application security support in the operating system kernel. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)*, pages 201–211, Taipei, Taiwan, March 2006.

[29] Manigandan Radhakrishnan and Jon A. Solworth. Quarantining untrusted entities: Dynamic sandboxing using LEAP. In *Annual Computer Security Applications Conference (ACSAC)*, page to appear. ACSA, December 2007.

[30] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[31] Ravi Sandhu, Mihir Bellare, and Ravi Ganesan. Password-enabled PKI: Virtual smartcards versus virtual soft tokens. In *Proceedings of the First Annual PKI Research Workshop*, April 2002.

[32] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[33] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[34] Donn Seely. A tour of the worm. Technical report, Department of Computer Science, University of Utah, 1988.

[35] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In USENIX, editor, *Proceedings of the Tenth USENIX Security Symposium, August 13–17, 2001, Washington, DC, USA*, pages 201–218. USENIX, 2001.

[36] Jonathan Shapiro. bitc. `http://www.bitc-lang.org`.

[37] Jonathan Shapiro. Programming language challenges in systems codes: why systems programmers still use c, and what to do about it. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 9, New York, NY, USA, 2006. ACM Press.

[38] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proc. IEEE Symp. Security and Privacy*, pages 166–176, 2000.

[39] Brian D. Snow. We need assurance! In *ACSAC*, pages 3–10, 2005.

[40] Jon A. Solworth. The PARSEQ project: An interim report. In *Languages and Compilers for Parallel Computing*, pages 490—510. Pittman/MIT, 1990.

[41] Jennifer G. Steiner, B. Clifford Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988.

[42] J. E. Thornton. *Design of a Computer: the CDC 6600*. Scott, Foresman & Co., Glenview, IL, 1970.

[43] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Research and Development*, 11(1):25–33, January 1967.

[44] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*, pages 203–216, 1993.

[45] C. Waldspurger. Memory resource management in VMware ESX server. In *Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[46] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *FAST*. USENIX, 2005.

[47] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, Washington, November 2006.