Executive Summary

Developing secure software is critical to a company’s reputation and bottom line. The impact of a software malfunction or security breach can result in a massive recall, millions in lost revenue, the loss of sensitive customer data, and a headline in the Wall Street Journal. With the ubiquity of software in everything from automobiles to investment trading applications, more companies than ever are feeling the legal and financial pressure to assure the security of their code. Faced with having to maintain software quality and security while accelerating innovation, companies with institutionalized, standard code development processes are looking for new ways to further reduce overall program risk. Traditionally, companies would perform security testing near the end of the software development lifecycle, prior to product release but that process can put release schedules at risk and late-found defects cost more to address. To more effectively address security, some of these companies are now adopting secure development lifecycle initiatives where security deliverables are inserted in all phases of development. As a result, companies are finding that the benefits of fewer security incidents, faster time to remediate and earlier visibility into areas of risk far outweigh the costs of implementing these initiatives.

So, are secure development lifecycle initiatives merely academic, or can they truly serve as practical guidelines for any but the largest companies?

This white paper outlines a practical approach to implementing secure practices into the software development lifecycle. And it is only by pushing past the traditional operational view of security and inserting it into all phases of development that you can begin to build software systems that can stand up under attack.
Software is a business necessity

Software is becoming embedded in nearly every device from power grids to automobiles to medical devices. In fact a term, The Internet of Things\textsuperscript{1}, was coined to describe the trend in which everyday products and devices are increasingly connected to the Internet via embedded software agents and sensors. According to the GSMA, a worldwide association of mobile operators and related companies, there are approximately 9 billion connected devices today. By 2020, the GSMA predicts there will be 24 billion connected devices in the world and over half will be non-mobile devices. These will include home devices, healthcare devices and automotive technology - all connected via software.

With its ubiquity, software has become the infrastructure that any sizable business depends upon to function on a daily basis. However, unlike concrete and steel infrastructures that gird the buildings in which these businesses are housed, it is not nearly as reliable or fault-tolerant.

Developing secure software is still a challenge

Developing reliable and secure software is a tough challenge that confronts IT teams – both security and development teams. Traditionally, computer science programs have focused on producing programmers with a foundation to become good application developers but not necessarily security experts. As a result, developers today are by and large unaware of the myriad ways they can introduce security problems into their code. There is also currently a misalignment between stakeholders across the software development life-cycle:

- Misaligned priorities - Development teams are focused on product innovation to meet business needs. Vulnerabilities stemming from code defects are seen as potential problems, therefore not a priority compared to feature functionality and on-time delivery. QA teams are concerned about buggy software and customer dissatisfaction. Security teams are focused on the availability and protection of sensitive assets – they are tasked with securing in-house and commercial applications, often having to address vulnerabilities exposed by software code after it is deployed.

- Misaligned process - Security audits and QA testing happen at the end of the development cycle where issues are most expensive to fix and when developers are focused on getting the release out and moving on to the next release. Audits are typically done late in the cycle to avoid having security experts review and re-review code that is likely to change before release. Also, security audits typically happen outside the standard development workflow, which means developers are likely to ignore security issues identified during the audit because it is hard to go back and change “working” code without causing an expensive and lengthy testing cycle. Therefore security issues identified late present business stakeholders a difficult decision between time to market and security.

- Misaligned tools - Developers resist changes to their workflow and find it difficult to use tools designed for security experts. They require too much security expertise and do not provide directly actionable information for fixing defects. Putting security auditing tools in the hands of a developer is not a practical solution as these tools are designed to find every possible issue resulting in a high false positive rate. Developers will often ignore the tools analysis results if they have to wade through a high volume of noise to identify critical defects that must be fixed.

Unfortunately as we know, all too often defects ignored or overlooked during the development process end up causing major issues down the road. Due to the proliferation of software, we are at that uncomfortable juncture in the history of software development where we are seeing firms suffer businesses disruption and negative publicity because they failed to manage risks posed by insecure code practices. Breaches as a result of application vulnerabilities have been reported across industry segments and geopolitical boundaries:
• On April 19, 2011, Sony learned that its PlayStation Network (PSN) had been hacked as a result of vulnerabilities in the application that made it susceptible to SQL injection type attacks. Three days later, reports appeared indicating that 2.2 million credit card numbers were being offered for sale on hacker forums. Two weeks later, the PSN was still unavailable, leaving many users not only with potentially compromised credit cards, but also without access to the games they had already purchased. All told, it is estimated that 77 million accounts were compromised, making it one of the largest data security breaches in history, surpassing the TJX hack in 2007 where an estimated 45 million customers were affected.

• On January 2011, hackers from the Anonymous group broke into a vulnerable HBGary Federal webserver through an SQL injection attack. HBGary Federal had been working with the FBI in an investigation into the attacks against companies who were cutting off WikiLeaks access and financing. By obtaining the passwords belonging to the CEO and COO, the attackers were able to get access to the company’s research documents and thousands of proprietary emails.

• In June 2011, an NSS-Labs security researcher also discovered vulnerabilities in supervisory-control-and-data-acquisition (SCADA) systems used in China which would have allowed denial-of-service attacks and remote code execution. In a scenario reminiscent of the Stuxnet attack on Iranian nuclear facilities, a successful exploitation of these vulnerabilities “can result in adverse application conditions and ultimately impact the production environment on which the SCADA system is used,” said the ICS-CERT advisory. The software used by the Chinese Sunway systems is also widely used in Europe and the Americas to control critical infrastructure in the petroleum production, petrochemical, defense, transportation, water, manufacturing and energy industries. Note: To date, ICS-CERT has coordinated with the China National Vulnerability Database and Sunway to resolve the problem.

Given the financial and legal ramifications as well as customer impact from the loss of sensitive data from breaches, it is apparent that software vendors have both a responsibility and a business incentive to ensure software security. A broader approach to software security is needed

When it comes to software development, security needs to be brought in from “around the edges” – it is only by pushing past the operational view of security that we can begin to build software systems that can stand up under attack. Security defects can, and should be treated like software defects and managed as part of the development process. Indeed the distinction between security and quality can sometimes be a subtle one; the bug that manifests as a system failure today could be exploited by an attacker tomorrow. For near-term progress on developing more secure software, the most important people to enlist are the builders of the code - the developers.

In conjunction with consortiums like SAFECode and BSIMM, companies like Microsoft, Cisco and Adobe have taken the lead to establish secure code development initiatives that inject a set of security deliverables into each phase of the software development process. The general focus of these initiatives is to establish a foundational set of practices, including training and engaging developers on security practices, aimed at reducing the number and severity of vulnerabilities in software.

The need to consider security and privacy “up front” is a fundamental aspect of secure system development. The optimal point to define trustworthiness requirements for a software project is during the initial planning stages. This early definition of requirements allows development teams to identify key milestones and deliverables, and permits the integration of security and privacy in a way that minimizes any disruption to plans and schedules.

Simplified Implementation of the Microsoft SDL
In general, these SDL models are structured around mapping security into key phases of the software development lifecycle:

- **Training, policy and organizational capabilities**
  - Training software development teams in the basics of security and ensure they stay informed of the latest trends in security and privacy

- **Planning and design**
  - Consider security and privacy in the initial design of new products and features and permits the integration of security in a way that minimizes disruptions to plans and schedules

- **Implementation**
  - Avoid coding issues that could lead to vulnerabilities and leverages state-of-the-art development tools to assist in building more secure code

- **Verification and testing**
  - Run-time verification of software applications to ensure that functionality works as designed.

- **Release and response**
  - Response plans and protocols to address new threats that emerge over time.

By introducing security early in the development lifecycle, companies are able to meet their customer demands for more secure products and services. And companies can derive additional benefits such as reduction in patch maintenance and faster time to remediate.

**Proven value for secure code development and testing early in the cycle**

It takes significant effort to determine the exact cause of defects and sometimes fixing defects can involve extensive architectural changes. Finding critical defects as early as possible enables companies to anticipate workload and impact to release schedules, thereby reducing cost to the overall project.

Microsoft reports that the adoption of the SDL process has led to a 50%-60% reduction in security defects⁴ that make it into production. The benefits according to Microsoft, of fixing a bug early on during development:

- Fixing a bug early avoids wasted time debugging strange crashes or reliability issues close to product release.
- Fixing a bug early avoids resetting/repeating testing after a bug is fixed late in the development cycle.
- Fixing a bug early avoids the complexities associated with fixing it if it is exposed after the application ships.
A recent study conducted by Coverity with 12 customers showed the following ROI in the areas of defect prevention, developer productivity and increased efficiency through automation:

- Average total annual savings of $600K from defect prevention for projects with 1 million lines of code.
- Average annual savings of $280K from increased developer productivity based on average time needed to fix a defect before and after using Coverity.
- Total automation savings of $2M based on average of 10-15 FTE that will be needed to review code manually to match Coverity’s coverage.

Detailed analyses and customer data can be obtained from The Proven ROI of Development Testing report.

Industry data also underscores the ROI gained from testing code early in development. According to the National Institute of Standards and Technology (NIST)\(^5\), the cost benefits of finding and addressing defects early are staggering. For every $1 spent on addressing defects during the coding phase of development, it will cost an organization $30 dollars to address if detected in production.

The clear takeaway here is that any kind of secure development lifecycle initiative represents good business value.

**A practical approach to secure code development\(^6\)**

Designed for developers, Coverity development testing solutions address the major issues that have made developers shy away from the earlier tools: usability, efficiency and false positive reporting. It also provides a practical solution for companies who may not have the resources to implement a full-fledged SDL program.

The Coverity Development Testing Platform provides development teams with a quick and easy way to test code for defects in a non-intrusive manner. It integrates seamlessly with IDEs like Eclipse or Visual Studio, so developers can quickly identify quality and security defects from within their IDE at their desktop, without disrupting their workflow. Once defects are found, developers are automatically notified of defects within their existing workflow, prioritized by risk and impact, so they know which problems to fix first. Developers have one-click access to a rich defect knowledge base which takes the guess work out researching unfamiliar defects, and helps developers more quickly identify the root cause of a defect. Because many organizations leverage shared code across projects and services, Coverity® Static Analysis will also show the development team all of the places across the shared code where that defect exists, so a fix can be applied in all these places.
Coverity Static Analysis is also available pre-configured for the Wind River Workbench, supporting both Wind River Linux and Wind River's VxWorks real-time operating system (RTOS). This combined solution enables development teams to build security into the embedded software development process and effectively address security vulnerabilities as software code is written.

As development teams have mission critical priorities besides security, and can only allocate a certain amount of time to fixing code, the slow uptake of analysis tools by developers has been due to the high volume of false positives making it daunting to sift through the thousands of defects churned out by these tools. The focus for Coverity from the beginning has been on providing the industry’s most accurate analysis solution and the lowest false positive rates averaging less than 15% and as low as 1%, enabling developers to work on real and relevant defects instead of wasting development cycles.

Bringing security into the development phase of the software development lifecycle puts the responsibility of fixing common, but easily overlooked critical code defects into the hands of the developer. And developers can play a critical role in helping security teams minimize attack risk by removing the conditions in software that make it vulnerable to exploits.
**Increased visibility to security posture across the entire development lifecycle**

In addition to analyzing code on their desktops, developers can benefit from a central build system or continuous integration server such as Jenkins, where the analysis engine can evaluate the cumulative changes of the entire development team.

**Coverity® Integrity Control** provides a visual representation of the areas of risk across projects and teams. A development manager or security auditor can view a hierarchical heat map that is tailored specifically to the needs of the organization. Distributed teams can be tracked to ensure they are executing consistently and cross-functional team leaders can work together to address any potential areas of risk and skills gaps. Coverity Integrity Control provides the visibility and control needed to consistently measure internal teams as well as suppliers against the same standards for quality and security – with the ability to audit SLA violations on-demand. Security engineers, defect teams and development team leads can drill down into each policy to pinpoint the full context of the code problem, identify the specific policy in violation and where it originated. Defects can quickly be assigned to the appropriate developers, facilitating the triage process. An updated risk profile report can be generated with every code iteration and test for audit purposes.

*Executive-level visibility into areas of risk*
Set policies for security, quality and efficiency based on business priorities

Coverity facilitates a common workflow for code testing between stakeholders that promotes consistent policies and controls across internally developed, open source, third party provider code; and increases visibility into composite application security risks.

Coverity Integrity Control lets users centrally define software development policies based upon business priorities, regardless of whether it is a quality or security issue. Once defined, the policies are centrally published and then shared with cross-functional and dispersed teams. Specific policies which can be defined include:

- **Security policies:** Users can establish and enforce policies for defects identified as security risks by the industry standard Common Weakness Enumeration (CWE), and establish policies for security defect density.

- **Quality policies:** Users can set policies for defect density, critical defects and uninspected defects. Critical defect policies can be established as well as thresholds for uninspected defects since these defects could represent a risk to the overall code quality.

- **R&D productivity:** Users can establish policies for internal teams and third-party suppliers for critical metrics tied to R&D efficiency.

- **Usage and savings policies:** Users can establish policies around the usage of Coverity Static Analysis and Coverity® Dynamic Analysis. Policies can be established for the number of active users, projects and lines of code being scanned. This can be critical in enforcing code quality and security across an organization and supply chain.

Mitigating risk early in the development lifecycle

Coverity Static Analysis identifies exploitation primitives, conditions in software that serve as the basis for exploitation. While a single primitive in and of itself may pose minimal risk exposure, multiple primitives can be chained together by attackers to gain control of an application. By eliminating as many primitives as possible, developers can make the process of exploiting software much more difficult for the attacker. This is akin to having a tool-kit with multiple tools for a specific home repair job. While a paint job can be completed with just paint and a brush; having a roller, a paint tray, painter’s tape, a putty knife and spreader makes the job much easier and faster.

Example of a multi-primitive attack

Let us assume that a security vulnerability exists in the code that resides on a remote server. While identifying the root cause of the security vulnerability is sufficient for remediating the flaw, the successful exploitation of that vulnerability is dependent on multiple pre-existing conditions. For the context of this example, we will assume that we have an attacker attempting to achieve remote code execution (RCE); thereby running code of the attacker's choosing on the remote machine. Although triggering the security vulnerability is required to achieve RCE, it actually requires many small steps that we refer to as exploitation primitives. By chaining these primitives together, the attacker can create an exploit that works reliably and maintains stability after the exploit has run its course.
In our example, the attacker is using but is not limited to four unique primitives. The first primitive used is the **soft leak**, which leverages legitimate program functionality to manipulate memory in the targeted application without any stability or security repercussions. These primitives happen to be the most common because they rely on intended, valid program functionality. For example, a server, by design, will accept requests from a client. That client sends information which is held until session termination occurs. By figuring out how these requests and sessions work, an exploit writer can make certain assumptions about the memory layout of a particular application based on its functionality.

The next primitive used is the **hard leak**. The hard leak, or resource leak, is quite familiar to most C/C++ programmers. The leak occurs when the programmer forgets to free memory that was acquired dynamically during runtime. While most programmers think of this as a quality problem that will result in massive memory consumption at worst, many exploitation artists see this as an opportunity to ensure exploit stability. By acquiring memory permanently, an attacker can assure that certain portions of memory are never subsequently used throughout the lifetime of a process.

The third primitive used is the **integer overflow**. If a mathematical operation occurs that attempts to store a number that is larger than an integer can hold then the excess is lost. The loss of the excess data is sometimes referred to as an integer "wrap". For example, an unsigned 32-bit integer can hold a maximum positive value. By adding 1 to that maximum positive value, the integer will start counting again at zero (UINT_MAX + 1 == 0). A real world example is the odometer of a car rolling over after 1 million miles; re-starting its mileage count from zero. By using this overflowed integer in an allocation routine, an attacker can allocate less memory that was intended.

Finally, the last primitive used is a **buffer overflow**. This is the most common kind of defect that is understood to have security implications in C/C++ programs. A buffer overflow is caused when the program writes past the end of a buffer, resulting in corruption of adjacent memory contents. In some instances, this may result in overwriting the contents of the stack or heap in ways that allow an attacker to subvert the normal operation of the system and, ultimately, take over the flow of control from the program.

Now that the primitive types have been covered, let us discuss how the attacker in our example utilized them to achieve remote code execution. First, by using existing program functionality, the attacker sends valid requests that result in allocating many chunks of memory based on the size of his input. This may seem harmless but is vital to achieving heap determinism, the manipulation of the memory layout of an application into a known desirable state, which is obligatory when exploiting heap-based buffer overflows. Next, the exploit author knew that some memory, once allocated, should never be freed again. By leveraging hard leaks within the application, the goal of having memory that survives throughout the life of the process can be achieved; resulting in greater post-exploitation stability.

The integer overflow that caused an under allocated heap buffer to be overflowed was triggered. This causes a mismatch between the actual size of the allocated buffer and the expected number of data elements it holds. The attacker can then leverage a buffer overflow to overwrite the contents of adjacent memory. For example, imagine you have the inability to determine the last line of a piece of ruled paper. If you sequentially keep writing sentences, you would eventually write onto your desk and potentially that nice new shirt! By overwriting adjacent memory, the attacker can overwrite important information with data that he controls.
There are many more details that involve different kinds of exploitation primitives, but a clear picture has been painted. Being able to chain primitives together, regardless of severity, results in a greater ability to control exploitation and post-exploitation functionality. If our attacker did not have the ability to create hard leaks within the application, he would have had to figure out a different way to ensure that his memory was not freed when his session timed out, or at least come to the realization that an eventual program crash was inevitable. And if the integer overflow did not exist, there would not have been an opportunity at all for our attacker to exploit.

As you can see, primitive chaining is a vital process during exploitation. As illustrated below, by limiting the number of primitives within code, developers can make the process of exploiting software much more difficult, thereby increasing the cost of exploitation.

Coverity Static Analysis identifies exploitation primitives, some of which may be vulnerabilities. Most vulnerabilities fall into the following categories:

- **Arbitrary code execution:**  
  Modern operating systems on most architectures intertwine code, data, and meta-data. Therefore, buffer overflows can result in the overwriting of important information used to control the flow of execution. Defects like copying of un-sanitized user-provided data into fixed-size stack buffers and dynamic allocation of buffers that are too small result in crashes and unexpected behavior at best, and altered control flow and unauthorized code execution at worst. Defects like integer and buffer overflows, use of previously freed pointers, and use of scalars (for example, integers) that are not properly bounds-checked (sanitized) before being used as array or pointer indexes, loop boundaries, or function arguments are possible primitives for code execution.

- **Data loss:**  
  Unlike unavailability, this severe class of security vulnerabilities results in data loss due to corruption, modification, or theft. Data loss occurs due to primitives that modify application data in memory or in persistent storage, or allow data leakage and seepage, and rather than affecting the control flow of execution, affects data and meta-data. Data loss features prominently in the 2011 CWE/SANS Top 25 Most Dangerous Software Errors list.
Security bypass:
An attacker can leverage software defects to circumvent authorization and authentication modules resulting in a security bypass. Defects in software where filenames are unsafely checked before being used leading to a file-based race condition can lead to unauthorized access to resources in extreme cases. Similarly, primitives that allow overwriting of state variables like “isAdministrator” or “isAuthenticated” typically used in the control flow are examples of security bypass primitives.

Denial of service:
This class of vulnerability manifests from defects that lead the software application to fail, to perform in an unintended or an undefined manner, or to slow down. Typically, defects that lead to crashes and unintended behavior also manifest into denial of services as the response of most software is to exit the application, thus preventing legitimate use. Defects like infinite loops and deadlocks are direct cases of denial of service vulnerabilities. Additionally, primitives can be combined to structure an attack such that it makes requests to allocate a specific resource (e.g. memory) that is not being freed. This results in resource exhaustion and a denial of service when a legitimate request for the resource is made.

Loss of integrity:
Integrity is defined as the expectation of reliability in terms of behavior and performance from a software application or process. Defects where the language rules for expression evaluation do not determine the order in which side effects happen, misuse and mixing of data types like enum are broadly classified as a class of errors where the integrity of the software is compromised. These vulnerabilities are typically the result of inadvertent errors in program code and not as easily exploitable.

The link between exploitation primitives and security vulnerabilities can be direct or indirect. Certain kinds of primitives, such as buffer overflows, can lead to many different kinds of vulnerabilities, depending on the skill, creativity, and determination of the attacker. What is clear, however, is that having more primitives available makes it easier for an attacker to leverage more severe vulnerabilities and develop damaging exploits. Therefore, finding and eliminating large numbers of exploitation primitives early in the development process can greatly help in reducing security vulnerability exposure and maintenance costs over the entire time the application is in service.

A comprehensive list of Coverity Static Analysis checkers for C/C++, C# and Java codebases are available on the Coverity website at www.coverity.com.

Securing the code of security technology companies
Coverity secures the software code of over 1,100 of the world’s leading companies, including three of the top five security software companies, to help them meet the highest levels of application quality and security and reduce the risk of costly brand damage.

A Fortune 500 security technology leader in authentication, encryption and access control
This company has established a comprehensive approach to secure software development that crosses people, processes and technology. The company created a product security policy to benchmark product security against market expectations and industry best practices. In addition, it actively promotes a security-aware culture across its engineering community, offering role-based security training for developers. Coverity is currently used by over 13 development
groups worldwide, analyzing more than 28 million lines of code (LOC). Developers use Coverity Static Analysis to analyze their work on a nightly basis before checking in their code. The security policy team certifies that the developer code is clean before it is released and incentivizes developers who capture security defects by giving them compliance “credits”.

A Fortune 500 leader in networking and network security

With annual revenues of over $10 billion, this technology leader has a company-wide secure development lifecycle process to increase the resiliency of their products. Automated static code analysis is seen as a core component in this process as IT management recognizes that quality defects are entry points to vulnerabilities. Looking for a solution with low false positive rates, they found the Coverity solution to be best in market and the IT division responsible for the operating system implemented Coverity Static Analysis to test the codebase for their operating system which is embedded in all their networking appliances. The operating system application had been vulnerable to buffer overflows and other problems. Since the operating system application had a monolithic architecture, which means it runs on a single image and since all processes share the same memory space, there is no memory protection between processes, so bugs in the operating system code can potentially corrupt data used by other processes. For the company’s products that required very high availability, these vulnerabilities were not acceptable.

To date, the company has implemented Coverity on a huge scale, with 2000 developers running Coverity Static Analysis on their desktop for incremental builds before check-in. This enables them to run their analysis against the baseline for efficient triage. As part of a version control process, each time an analysis is run, a Coverity Static Analysis report is generated, providing a record for the clean before check-in audit. A full analysis of the application codebase is also done periodically in order to catch any defects that would surface only during the full context analysis, particularly as part of a statistical-based analysis. Profitability and time savings equated to over 24,000 man days saved, resulting in $11M savings per year.

Software company rated among top 5 most valuable brands in the world

Concerned about the quality of software coming out of their worldwide development centers, this company approached Coverity. During a vendor product bakeoff, Coverity Static Analysis was able to catch 80-85% of the C/C++ flaws that were considered critical defects for this company. Coverity is currently used by the corporate security audit team which does internal audits on different C/C++ and Java code bases (up to 25 million LOC) for production level quality and readiness for shipment. The team realized a savings of 214 man hours per week after using Coverity Static Analysis.

A leader in email and mobile security, one of the world’s largest dedicated security technology companies

A Coverity customer for 7 years, this company rolled out Coverity static analysis to around 400 developers worldwide at their main development centers in US, Europe and Asia. As the company has grown by acquisition, their distributed development teams have been required to run Coverity as a gate for quality and security for C/C++ and Java codebases that now total over 35 million LOC.
Conclusion

Defects are an inevitable fact of software development. While it might not be possible to completely prevent vulnerabilities from being introduced during code development, especially for large-scale projects, the technology and processes exist now to assist developers in finding and fixing these defects as quickly and efficiently as possible.

However, technology adoption in development will be limited unless it is effortless for the developer. If testing results are not presented to the developer in a way that is actionable, relevant, and integrated into their workflow, the troubleshooting and fixing of defects will slow down the entire process or the technology may simply become "shelfware". Given the time to market pressure on development, defects will likely go unresolved, or go undetected until it is too late and a customer or exploit artist discovers them.

Coverity provides the industry’s first developer friendly and enterprise-ready development testing platform, allowing development organizations to seamlessly integrate code testing into the development process. Used in tandem with secure development practices, Coverity development testing solutions train developers to address both security and quality when testing code. By empowering developers to fix the commonly found but potentially critical defects, Coverity takes some of the burden off of over-extended security and audit teams and enables them to better focus their resources on the overall system level security.

Secure development testing also affords security teams with higher visibility into security risks early in the product lifecycle. It serves as a natural complement to traditional security testing methods such as vulnerability scanning and penetration testing, providing security teams with another tool in their arsenal against constantly morphing threats.

Citations

2 2010. Simplified Implementation of the Microsoft SDL.

For More Information

Find out how Coverity can help your organization improve the quality and security of your software and how it can be integrated into your software development lifecycle. Contact your Coverity representative or visit us at www.coverity.com.