Inside the Windows Security Push: A Twenty-Year Retrospective

Steve Lipner | SAFECode
Michael Howard | Microsoft

This article is a follow-up to an article on the Windows security push in the first issue of *IEEE Security and Privacy* (January 2003). It provides additional detail on the security push and its results, and describes the creation and evolution of the Security Development Lifecycle (SDL) that integrated software security into Microsoft's development process. The article concludes with a summary of lessons learned about effective ways of creating secure software at scale.

In the first issue of *IEEE Security & Privacy* (January 2003), we published an article entitled "Inside the Windows Security Push."[1] The article described some aspects of an intense effort by the entire Windows Division at Microsoft to improve the security of the next release of the Windows Server software. The security push was conducted in early 2002 and the development of the Windows Server release was still underway when the article was published.

The editors of *IEEE Security & Privacy* asked us to revisit the article and the security push effort on the occasion of the magazine's 20th anniversary. In this article, we'll amplify some points in the original article and correct a few errors. We'll also trace the impact that the security push had on secure development practices at Microsoft through 2015 (when Lipner retired from Microsoft) and across the software industry to the present day. Finally, we'll summarize key lessons learned that are important to organizations that seek to deliver secure software to their customers.

Revisiting the Security Push

Origins

The security push was launched at a time when Microsoft was undergoing something of a software security crisis. In late 2000 intruders gained access to the Microsoft corporate network. The intrusion was made public and raised concerns by customers as well as the U.S. Government. The summer of 2001 saw the release of the major Code Red and Nimda Internet worms. While the intrusion did not result from exploitation of vulnerabilities in Microsoft software, the worms did, further magnifying customer concerns and pressure on Microsoft management to "do something."

In 2001, Craig Mundie, then a Microsoft senior vice president and chief technical officer, and the late Howard Schmidt, then chief information security officer, began to advocate within Microsoft for expanded attention to product security. The Secure Windows Initiative (SWI) team in the Windows development organization, which had been formed in early 1999, began to emphasize security training for developers in the Windows Division and across the company, and the Microsoft Security Response Center (MSRC—Microsoft's Product Security Incident Response Team) stepped up its efforts to release timely fixes for reported vulnerabilities. (Over the last 20 years, the name of the team at Microsoft that creates and manages secure development processes has changed several times. For simplicity, this article stays with the original name—the Secure Windows Initiative or SWI).

By the fall of 2001, it was clear that Microsoft was not making enough progress. Vulnerability reports continued to arrive and were often released to the public

Digital Object Identifier 10.1109/MSEC.2022.3228098
Date of current version: 10 April 2023

 Copublished by the IEEE Computer and Reliability Societies

rather than being reported privately so that Microsoft could release fixes to protect customers from exploitation. Windows XP, a new release that had undergone SWI reviews and some developer security training, seemed to have no better security than prior versions.

In early November, a brainstorming effort at a joint offsite meeting of the SWI and MSRC leaders surfaced the ongoing .Net Framework security push and raised the question "could we do that for Windows?" One of us (Lipner) raised that idea in a meeting with the manager whose team included the SWI and MSRC. The response was that the idea seemed half baked (it was) and that there was great concern about whether the engineers in the Windows Division would take the effort seriously. ("They'll just roll their eyes.") But that meeting launched a two month-long planning effort that nailed down many of the details described in the original security push article: training sessions, running the then-available static analysis tools, signoff on every source file, relying on the techniques documented in *Writing Secure Code*.[2] Michael and David LeBlanc wrote *Writing Secure Code* as a response to common security questions from engineering teams. They wanted to focus on hard problems rather than day-to-day minutiae, so they wrote down the elements of secure design, coding, and test. The book proved to be a great teaching resource and reference during the security push.

Through December 2001 and January 2002, we continued to plan, create training materials, and brief progressively higher levels of management. At some point in mid-December, a senior executive asked what dates he should reserve the 1,000-person meeting room in the Microsoft Conference Center for the training sessions, and at that point we concluded that the security push was approved.

While the security push planning was going on, discussions of a corporate commitment to product security had continued. They culminated in mid-January with Bill Gates' release of the Trustworthy Computing (TwC) e-mail.[3] While the TwC e-mail and the security push were pursued somewhat independently, they were synergistic: The TwC e-mail was a major factor in causing developers to take the security push seriously, and the security push was a major factor in convincing doubters inside and outside the company that Microsoft was serious about improving security.

Execution

The security push began in late January 2002 with training for every engineer working on the new Windows release. Over a period of five days, the SWI team delivered ten four-hour training sessions to groups of around 900 engineers. (The original article refers to ten days of training, but that is an error.) Each training session was introduced by a vice president from one of the teams in the Windows Division. Any reader who has worked in a large organization knows how this normally works—the vice president stands up and says, "this is really important; pay attention," then leaves the room. In the case of the security push, the vice presidents gave their introduction then sat down and stayed for the full four-hour training. We have always believed that detail, like the TwC e-mail, was a major factor in the developers' belief that Microsoft was serious and their commitment to the security push.

One aspect of the security push that we didn't adequately emphasize in the original article was the management of the ongoing process. With 8,500 developers, the Windows Division had an established process for tracking progress, approving changes, and ensuring that Windows continued to progress against its release goals. The Windows Division leadership managed the security push as a normal Windows development activity. A shiproom ("war team") meeting each morning reviewed statuses, assigned problems for action, and reviewed changes. While changes to make code conform to the guidance in the training and *Writing Secure Code* or to fix errors discovered by static analysis or testing were automatically approved, design changes such as decisions to remove components or disable them by default were discussed at the morning's shiproom meeting. The progress and achievements of the security push were the primary focus of the weekly Friday-afternoon all-hands meetings for the entire Windows Division, and the executive who led the division handed out cash prizes for the "best bug" of the proceeding week. An oft-cited favorite "best bug" was the "oldest bug written by the most senior person".

Not everything went smoothly during the security push. Threat modeling was in its early days in 2002, and our guidance and training were only actionable by experts at software security. One of us remembers a development lead (who later became responsible for Microsoft's product and online service security efforts) becoming frustrated and angry with the difficulty of following the guidance about threat modeling. The security push did produce threat models and used them, but the best were developed with the aid of SWI team members or of some consultants who were on site during the push. The following sections mention the evolution of threat modeling after the security push.

One useful precedent that the security push set was the introduction of code-level mitigations. Mitigations serve to reduce the severity or exploitability of vulnerabilities that tools and code reviews miss. For example, Visual C++'s stack-based memory corruption detection flag, -GS, was mandated. For Windows Server 2003, this was the only security-related compiler flag we had, but

over the years many more would follow. We also started to focus on banning certain types of potentially dangerous C functions, including strcpy(), strncpy, sprintf(), get(), and many more. (A copy of the C header that bans potentially dangerous C runtime functions is available at Michael's GitHub repo.[4])

The original plan for the security push had been to review both the Windows Server version in development and Windows XP (which had shipped in August 2001). The modified code would ship as a new release of Windows Server and as Service Pack 1 for Windows XP. The plan to include Windows XP in the security push was dropped midway through the push for three reasons:

1. There was concern that customers would be unwilling to install a service pack that made the number of changes that the security push planned to make.
2. There was concern that some of the feature changes (e.g., removing features or disabling them by default) would have a negative impact on customers who were using Windows XP.
3. Microsoft's antitrust settlement with the U.S. Department of Justice required changes to the Windows default browser option by a specific date. That date was incompatible with the planned release date of the codebase that was going through the security push.

We included a small number of the most important code security fixes in Windows XP Service Pack 1, and it shipped well before the Server release. But we later delayed the Windows development process to ship a security-focused Service Pack 2 in 2004 and then had to delay the following desktop operating system release (Windows Vista) to transition the desktop product to the more secure Server codebase.

Late in the security push, the Windows team decided to restrict the default functionality of the browser (Internet Explorer or IE) drastically. Internet browsers were at that time (and remain) a rich source of product vulnerabilities and of risk to users. While the browser team did participate in the security push, the product team believed that the risks posed by using a browser on a server remained high and browsing Internet websites is not a primary usage scenario for a server. The team restricted the "IE Enhanced Security Configuration" significantly enough so that an administrator could use the browser for downloading software updates but not much else, and as a result that configuration was resilient to many common browser attacks.

The testing activities during the security push were not a penetration test as such, but they did discover many vulnerabilities and some new classes of vulnerabilities. The magnitude of the changes made by the security push was significant: In addition to reviewing all Windows components (tens of millions of lines of code), the team made several thousand code changes to eliminate real or potential vulnerabilities. The developers were instructed not to try to decide whether code that didn't conform to the guidance in *Writing Secure Code* was an actual vulnerability, but, as Michael repeatedly said, to "just fix it." In our experience, such determinations were more often wrong than right and took more time and effort than making and testing the code change to eliminate the suspicious code.

After the end of the formal security push at the end of March 2002, the Windows team returned to a more normal set of development activities. Those activities included completing feature modifications that had been specified and designed during the security push but not implemented and testing the product to ensure that the security push modifications had not compromised reliability, application compatibility, or performance.

Results

The security push release—Windows Server 2003—shipped in the early spring of 2003, a few months after the original article was published. Customers reacted positively to the release, and it did well in standing up to the perpetual onslaught of vulnerability reports. There was no customer pushback against the restricted browser configuration, and customers appeared to appreciate this example of reducing the system's attack surface.

In the summer of 2003, the Blaster worm was loosed on the Internet. The class of coding error exploited by the worm was called out for remediation as part of the security push, but a reviewer missed the instance of vulnerable code that the worm exploited. (There was then no tool that would flag that code pattern automatically.) Fortunately, the -GS mitigation prevented the worm from infecting an exposed system—the system would crash but not propagate the worm. The team felt that this result vindicated both the security push approach in general and the commitment to introducing mitigations that would prevent exploitation of remaining vulnerabilities.

As the original article mentions, other major Microsoft products went through security pushes of their own after the Windows push had wound down. The target versions of those security pushes shipped as new releases in some cases and as service packs in others. The SQL Server experience was particularly noteworthy: The SQL team did their push on SQL Server 2000, Service Pack 2 and shipped it in January 2003—the same week as the release of the very damaging Slammer worm. Unfortunately, few SQL Server admins had installed the service pack prior to the worm's release, and the impact of the worm was considerable. But as Service Pack 2 became the dominant version of SQL

Server in the field, we found that the rate of vulnerability reports also dropped precipitously—another validation of the security push approach.

Creating the SDL

No one at Microsoft believed that the security pushes had solved the company's security problems, and the Slammer (January 2003) and Blaster (July 2003) worms reinforced the view that software security would be an ongoing challenge and that more work was needed to live up to the TwC commitment. Jon DeVaan, the engineering executive responsible for customer satisfaction initiatives, asked Lipner to lead a cross-product group team to consider ways to improve customer satisfaction with product security.

At a high level, the solution to the customer satisfaction problem was "fewer vulnerabilities, no worms, less and better patching" but the working group drilled into what specifically could be done and how the SWI team, which was gradually being augmented with more developers and program managers, could contribute. Through late 2002 and most of 2003, the SWI team focused on supporting security pushes, delivering security training, and building its capability to discover vulnerabilities without waiting for external researchers to report them. The team also participated in the planning and engineering of Windows XP Service Pack 2, in particular playing a key role in the introduction of Data Execute Protection, a mitigation for memory corruption vulnerabilities made possible by the introduction of the hardware No eXecute ("NX") technology[5] in new processors.

In late 2003, Craig Mundie raised the need for a more systematic and widespread commitment to software security. Lipner followed up with a series of meetings with the customer satisfaction team and with product group executives to discuss the idea of a mandatory process and get feedback on what would work. The socialization of that idea culminated in a presentation by Lipner and Mike Nash (the vice president responsible for the SWI team) to CEO Steve Ballmer and his staff. The presentation proposed a "security development lifecycle" or SDL that would incorporate mandatory security training for engineers and a set of mandatory technical requirements that products would have to meet before they shipped. The proposal included the notion that the SDL would be updated over time as threats and secure development techniques evolved. Ballmer's response at the end of the briefing was, "that's approved; we're not ever going to talk about it again."

The authors and Eric Bidstrup immediately began to document the requirements of the initial SDL version. Key elements of the SDL, in addition to annual developer training, were threat modeling, use of secure coding techniques, static analysis, enabling mitigations, security by default, tracking of imported components, and a security push. The SWI team would oversee a "final security review" or FSR that verified that the product team had met the SDL requirements and could include a penetration test of the finished product.

The initial version of the SDL (Version 2.0—we felt that the security pushes had been a de facto v 1.0) was detailed in a Word document and became mandatory on 1 July 2004.

Executing and Updating the SDL

Once the SDL mandate became effective, the SWI team immediately began to work with product groups to help them meet the SDL requirements, and to ensure that they had done so. Part of this role involved delivering in-person training to product group engineers. (This task was a major consumer of Michael's time during the early days of the SDL.) Because of our experience going back to the Windows security push and before, the team was well-prepared for this task although ill-prepared for the number of people who had to be trained. (During the presentation to gain approval for the SDL, Steve Ballmer remarked to Lipner and Nash, "you guys don't have any idea how many engineers this company has.")

Working with Product Groups

Helping product groups meet the SDL requirements and ensuring they'd done so was sometimes a challenge. While any group that had conducted a security push was well positioned to meet the SDL requirements, teams new to the process could present surprises. A couple of examples from the SDL's early days illustrate this:

- One product underwent a penetration test and was shown to be riddled with vulnerabilities. We determined that it wasn't fit to ship, and rather than tell the team merely to fix the vulnerabilities, we insisted that they delay shipping, go back, and actually execute the activities that the SDL required. When they returned for a second FSR, their new penetration test was clean—it was clear that they'd actually done the SDL. We asked the product group to fix the one or two vulnerabilities found and approved the product for release.
- A product that had come to Microsoft through an acquisition went through the SDL. As the SWI team member assigned to the product worked with the product group, it became evident that the product's architecture was fundamentally flawed: any attacker who turned their attention to a system running the product would be able to "own" it. The vulnerable product was already in customers' hands so telling the product group not to ship would not have improved the situation. We (Lipner and Matt Thomlinson who

then oversaw the program management team that worked with product groups) met with the product group executive and described the situation. We asked that the executive make a set of changes that would slightly mitigate the product's problems and that he commit to making the major changes to eliminate the vulnerable architecture in a future release. The executive agreed, the product shipped with the modest changes, and a few years later the product group shipped a major revision with a secure architecture. We got lucky—vulnerability researchers never discovered or exploited the vulnerable architecture.

Like the security pushes, the SDL was primarily a product group responsibility. The SWI team would normally assign a single program manager as security advisor to work with a handful of smaller products or one or two major products. The program manager could assess compliance with most SDL requirements by running a tool or by executing a query against the product's bug tracking system. Only threat models stood out as requiring more in-depth review.

Of course, the fact that the SDL was a product group responsibility meant that product groups had to assign engineers to assuring that the team understood and met each of the SDL requirements. For a smaller product, one engineer might perform this task, while larger products created groups to provide training local to the group, help with threat models and make sure that tools were run and bugs fixed. (In the case of Windows, the security assurance group comprised tens of engineers.) These product team security groups comprise the "satellite" as detailed in the Building Security In Maturity Model research.[6]

Early in the evolution of the SDL, the SWI team took the position that "we can stop you from shipping." After a few years, at the suggestion of Scott Charney who succeeded Nash, we changed to a risk acceptance process in which the product group could approve shipping with exceptions to the SDL, but only after a joint review with the SWI team. Shipping with a major ("critical") vulnerability required a review meeting between the product group vice president and the vice president who oversaw the SDL. Less serious exceptions were reviewed by lower levels of management from the respective teams. In practice, if an exception reached a product group vice president, their reaction was almost always "what do you mean you can't fix that?" Approval of serious exceptions was rare.

Updating the SDL

When we sought approval to mandate the SDL in 2004, we made it clear that the specific requirements would evolve with the security landscape. The requirements in SDL Version 2.0 were uncontroversial—they were basically the security push requirements that many teams had been exposed to. One question that we had to confront early in the life of the SDL was "how will we decide on changes?" Jon DeVaan provided useful advice encouraging us to socialize changes with the product groups to avoid a product group rebellion or passive refusal to comply.

We recast the product security customer satisfaction team as the SDL Advisory Board and reviewed proposed changes and issues with SDL requirements with the board at a few points during each SDL update cycle. (Update cycles varied from a minimum of six months to a maximum of 21 months, with most updates annual.) When we issued the first update to the SDL, we had to decide on a transition rule so that product groups would know which version of the SDL was mandatory. We decided that a product would be held to the SDL version in place when development started or to the latest SDL version that had been in place for a year or more when the product shipped, whichever was newer.

We treated the SDL updates as though they were product releases with the product teams as our customers: Each update, whether a new process requirement or a new tool or both, went through a requirements review and a beta test. Proposed requirements were dropped if they were insufficiently executable by the product teams and tools were dropped if they were unreliable or generated too many false positives. Here too, we applied a risk management approach: If a class of vulnerability was very serious and it was clear that external security researchers or malicious attackers were focusing on it, that might justify adding an immature requirement or tool to the SDL. This was the case if the tool or requirement was the best we had, and the alternative was responding to a deluge of vulnerability reports with a deluge of patches.

We created new SDL requirements and tools mainly in adherence to Rick Proto's[7] dictum that "theories of security come from theories of insecurity." We analyzed new vulnerability reports in Microsoft and non-Microsoft products to see if they exposed new classes of vulnerability, and if so, we looked for an opportunity to introduce a new requirement or tool to eliminate vulnerabilities of that class or a mitigation to reduce their severity. The lesson of the Blaster vulnerability in Windows Server 2003 taught us to prefer giving developers tools over telling developers to look for problems: Even the most diligent and dedicated developer could easily overlook an error when reviewing a code base as large as Windows, Office, or SQL Server.

We took advantage of new security technologies that could make the SDL more effective. For example, as fuzz testing became a mature vulnerability hunting technique, we added fuzz testing requirements (and tools) to the SDL—for files and for network interfaces.

Evolving Requirements and Tools

Experience with Windows Server 2003 and Windows XP showed us that code-level mitigations were a worthwhile investment, and Microsoft continues to invest in mitigations to this day. Engineers in the SWI team, usually collaborating with engineers in the Windows and compiler groups, devised new approaches to mitigating stack and heap overflow attacks and attacks that relied on exception handling weaknesses to execute malicious code, among others. When mitigations were acceptable (effective, compatible with existing applications, and had adequate performance), we made it mandatory for product groups to implement them in new versions. Some mitigations crossed compiler, analysis tools and operating system—for example, mitigation and detection of heap corruption attacks. Advances in vulnerability research have sometimes enabled the defeat of a mitigation—this happened in some scenarios with the -GS compiler option—but our overall experience is that mitigations have been effective at raising the cost and reducing the effectiveness of attacks.

Beginning with the Windows security push, we took the position that product features should only be enabled by default if most users would need them, that network ports should be blocked by default, and that features (or products!) should not require privilege to run. We built and mandated tools to enable product groups and SWI team security advisors to confirm that these requirements had been met.

We were acutely aware of the ways that product groups who chose to implement encryption algorithms themselves or to introduce new kinds of encryption features could go badly wrong. The SDL specified in detail what encryption algorithms were approved and for what purposes. We eventually established the "Crypto Board" composed of product group, Microsoft Research, and SWI team encryption experts to review and advise on new ways that products sought to use or implement encryption.

Even before the Windows security push, the MSRC team had faced the problem of vulnerabilities in third-party components that products incorporated but that were developed by other product groups, or outside of Microsoft. We required each product to maintain a list of components (we used the old Digital Equipment term *giblets*) so that it could remediate vulnerabilities when they were discovered. The current push for software bills of material (SBOMs)[8] reflects awareness of this 20-year-old concern by industry and government.

In response to the clear problems with threat modeling, Shawn Hernan led the way in developing a structured process that told developers how to analyze a data flow diagram for threats (potential vulnerabilities). We documented Hernan's approach in Chapter 9 of the authors' book[9] on the SDL and built a tool that automated the threat model analysis and bookkeeping for developers.[10] The tool and process were first used at scale on Windows 7 with good success. While threat models were mandatory beginning with SDL Version 2.0, we did not mandate the tool: Development groups that had the expertise to threat model successfully using a whiteboard and Word document were free to continue doing so. Even today, Microsoft does not mandate a single threat modeling tool.

During the decade after the security push, several new trends arose that changed the way software was built, deployed, and used. Online services began to supplement or supplant on-premises "boxed products," and, with the growth of online services, rapid or continuous development models replaced multiyear development cycles. We made our first attempt at creating a version of the SDL for agile development in 2005–2006, and Bryan Sullivan created a successful model a few years later.[11] The key was to recognize that the SDL specifies required developer activities but that when they are performed can vary depending on the development model. If developers are deploying to production several times each day, their static analysis tool needs to be lightweight enough to give them actionable feedback before they deploy. The Microsoft development groups that moved to agile found Bryan's approach effective and consistent with their desired approaches to development and deployment.

We sought to ease the management and tracking of the SDL both for the SWI team and for the product groups. We created a tracking tool that enabled product groups to register a new release, determine what SDL requirements applied to their version, and then upload tool outputs or enter manual attestations that demonstrated that the product version met the requirements. The tool started its life as a "hack" on a PC under a program manager's desk and its functionality was enhanced over the years. Our long-term goal was to enable development groups to manage their SDL work using their workflow (bug tracking) systems rather than having to work in a separate system dedicated to SDL compliance. Today, most teams at Microsoft follow their SDL requirements using tools like Azure DevOps, and the results roll up to an overall company-wide process.

During the first ten years of the SDL, our task was made easier by the fact that almost all Microsoft products were developed in one of a small number of languages (C, C++, C#) and used Microsoft development tools. We created SDL requirements for products that ran on the Macintosh, which represented a significant code base and market. While some tools had to be ported by the SDL team, other tools were available on the Mac and their use was consistent with the intent of the SDL.

Sharing the SDL

We decided early in the life of the SDL that we would be relatively public about what we were doing. We did this both to build customer confidence in our commitment to security and TwC and to encourage companies who wrote code for Windows to address security so that our customers would have a more secure environment—a customer whose Windows system is attacked is not likely to distinguish between Microsoft code and third-party applications.

The authors wrote a book[9] on the SDL in late 2005, basing the content on the then-current SDL Version 2.2. We also held a workshop on secure development and the SDL for original equipment manufacturers (OEMs) who were to ship PCs loaded with Windows Vista. (We had discovered that many OEMs shipped PCs with the Windows file system configured so that all users had full read-write access to every file—including the operating system—undoing the carefully thought-out default file protections that the Windows group had designed.) We established an informal program to share the SDL, including training, process guidance, and some tools, with major independent software vendors who expressed an interest—Adobe and Cisco were two active participants who have acknowledged their participation in that program and its value. We worked with Microsoft Consulting Services (MCS) to create a consulting practice that would help customers create their own SDL processes and integrated the SDL into the processes that MCS uses to create software for customers. And we eventually shared relatively complete versions of the full SDL (sanitized of internal references)[12]

In 2007, Microsoft and several other companies that had created or were creating software security programs established SAFECode, a nonprofit organization devoted to allowing its members to exchange insights and ideas on creating, improving, and promoting scalable and effective software security programs. In the 15 years since SAFECode was formed, it has released guidance documents and blogs on a variety of topics including secure development processes, use of third-party components, threat modeling, fuzz testing, and the role of security champions in development groups.[13] SAFECode members also share approaches to emerging problems to help the members improve the quality and effectiveness of their software security programs.

Both SAFECode and Microsoft have worked with governments to share approaches to software security. In 2004, Microsoft initiated an effort to infuse SDL concepts into the international Common Criteria for Information Technology Security. In the end, the complexity of evaluating products against SDL requirements was too great for the Common Criteria governments to accept.

More recently, SAFECode collaborated with the National Institute of Standards and Technology (NIST) and BSA, The Software Alliance on the creation of NIST's Secure Software Development Framework (SSDF), Special Publication 800-218.[14] The U.S. Government's 2021 May Executive Order on Improving the Nation's Cybersecurity (E.O. 14028)[15] aligns closely with the SSDF and the list of secure development requirements in Section 4 of the Executive Order (Enhancing Software Supply Chain Security) will look very familiar to anyone who has studied or worked with the SDL.

Lessons Learned

Twenty years after the Windows security push and almost 19 years after the creation of the SDL, a few clear lessons stand out. While development lifecycle and deployment models have changed, new programming languages have proliferated, and the use of third-party and open source components has grown, these common threads are still important to any organization that wants to deliver secure software to its users:

- The product group is responsible for executing the secure development process and delivering secure software. The security team is responsible for giving the product groups the training, tools, and processes that enable them to deliver secure software successfully. If the security team attempts to "do security" for the developers, they will always be too late and the software that actually ships will never be secure.
- The evidence of secure development is the code, the bugs in the workflow system, the outputs of security tools, the giblet database (SBOM), and the threat models. All are artifacts of building secure software. If a security team insists on being provided with compliance documentation, they will inevitably be misled by the distance between the actual evidence and the documentation.
- "Theories of security come from theories of insecurity." Root cause analysis of vulnerabilities, whether found by an internal security team or external vulnerability researchers, is the key input to the creation and updating of secure development process and tools.
- Secure development is a quality process like modern manufacturing and continuous improvement results from root cause analysis and response to defects. Perfect software security is likely not achievable, and certainly not in viable commercial products. But secure development processes and continuous improvement have been shown to reduce vulnerability populations and raise the difficulty of finding and exploiting vulnerabilities.[16]
- Mitigations are an important aspect of software security that complement the quest to build more

secure software. They are worth exploring and investing in. ■

Acknowledgment

The success of the Windows security push and its evolution into the SDL were the result of hard work and commitment to software security by a lot of people over the last two decades. We are grateful to the members of the Secure Windows Initiative team and its successors for their work in creating and implementing the security push and the SDL. Product team engineers and executives demonstrated commitment to delivering secure software, willingness to learn and adopt new tools and techniques, and provided valuable suggestions and feedback. Microsoft customers encouraged our efforts and independent and academic security researchers created new techniques that we integrated into the SDL as it evolved. We have mentioned the names of a few key contributors in this paper, but we've undoubtedly omitted some important names. We thank you all.

The content of this article represents the perspective of the authors and does not represent the position of their organizations.

References

1. M. Howard and S. Lipner, "Inside the windows security push," *IEEE Security Privacy*, vol. 1, no. 1, pp. 57–61, Jan./Feb. 2003, doi: 10.1109/MSECP.2003.1176996.
2. M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA, USA: Microsoft, 2001.
3. B. Gates, "Bill gates: Trustworthy computing," *Wired*, Jan. 17, 2002. Accessed: Dec. 19, 2022. [Online]. Available: https://www.wired.com/2002/01/bill-gates-trustworthy-computing/
4. M. Howard. "Banned." GitHub. Accessed: Dec. 1, 2022. [Online]. Available: https://github.com/x509cert/banned
5. "NX bit." Wikipedia. Accessed Dec. 19, 2022. [Online]. Available: https://en.wikipedia.org/wiki/NX_bit
6. "Glossary of software security terms." BSIMM. Accessed: Dec. 19, 2022. [Online]. Available: https://www.bsimm.com/about/glossary.html
7. R. C. Proto. "Richard Proto." Wikipedia. Accessed: Dec. 19, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Richard_Proto
8. "Software bill of materials," Cybersecurity and Infrastructure Security Agency, Washington, DC, USA, 2022. Accessed: Dec. 21, 2022. [Online]. Available: https://www.cisa.gov/sbom
9. M. Howard and S. Lipner, *The Security Development Lifecycle*. Redmond, WA, USA: Microsoft, 2006.
10. Microsoft Azure Product Documentation, "Microsoft threat modeling tool," Microsoft Corp., Redmond, WA, USA, Aug. 2022. Accessed: Dec. 1, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool
11. B. Sullivan, "Announcing SDL for agile development methodologies," Microsoft Corp., Redmond, WA, USA, Nov. 10, 2009. Accessed: Dec. 20, 2022. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2009/11/10/announcing-sdl-for-agile-development-methodologies/
12. SDL Team, "Microsoft security development lifecycle (SDL) process guidance - Version 5.2," Microsoft Corp., Redmond, WA, USA, May 23, 2012. Accessed: Dec. 20, 2022. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=29884
13. *SAFECode Home Page*. (Nov. 6, 2022). SAFECode. Accessed: Dec. 20, 2022. [Online]. Available: www.safecode.org
14. M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities," NIST, Gaithersburg, MD, USA, Feb. 2022. Accessed: Dec. 20, 2022. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-218/final
15. J. R. J. Biden, "Executive order on improving the Nation's cybersecurity," White House, Washington, DC, USA, May 12, 2021. Accessed: Dec. 20, 2022. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
16. M. Miller. *BlueHat Israel 2019 - Matt Miller -Trends, Challenges, and Strategic Shifts*. (Feb. 2019). [Online Video]. Accessed: Dec. 20, 2022. Available: https://www.youtube.com/watch?v=PjbGojjnBZQ

Steve Lipner is the executive director of SAFECode, Seattle, WA 98122 USA, an adjunct professor of computer science at Carnegie Mellon University, Pittsburgh, PA 15213 USA, and chair of the U.S. Government's Information Security and Privacy Advisory Board. His research interests include software security assurance, other aspects of system security, and the approaches that enable organizations to create effective security programs and products. Lipner holds an SM in civil engineering from the Massachusetts Institute of Technology. He was elected to the National Academy of Engineering in 2017. Contact him at lipner@outlook.com.

Michael Howard is a principal security program manager in the Azure Data Platform team at Microsoft Corporation, Austin, TX 78613 USA. His research focuses on streamlining secure software development practices and the use of artificial intelligence in security. Contact him at mikehow@microsoft.com.