DZone
A DEVADA MEDIA PROPERTY

TREND REPORT

Application Security

BROUGHT TO YOU IN PARTNERSHIP WITH

STORJ | DCS

Welcome Letter

By John Vester, Technical Architect at CleanSlate Technology Group

While analyzing the changes provided in a recent merge-request, I noticed the developer submitting the changes for review overlooked a crucial aspect of the new API: the expected security was missing. When I asked the developer about the missing security-related logic, I was provided an answer something along the lines of "it is not my job."

I was stunned to hear such a response but quickly realized an enrichment opportunity had been presented before me, for this developer was hyper-focused on meeting the needs noted in the feature request driving the merge-request. Unexpectedly, the acceptance criteria failed to mention any security expectations. It was an oversight.

The conclusion I shared during our discussion was that "the security and the integrity of the application is the responsibility for every single team member." My primary justification for my perspective was for this very scenario — where the product owner failed to note the security in the acceptance criteria.

I am proud to conclude that this experience helped the developer become one of the most-trusted feature team members I have worked with while building RESTful APIs.

As a young college student, I became aware of the thoughts of economist Karl Marx. The biggest takeaway that I still maintain from his wisdom is that "if every person works for the common good of society, then the class struggle is theoretically gone."

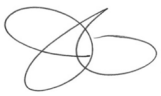
I have the same view on application security:

> *"If every team member maintains an eye on application security, there will be a natural improvement with the overall integrity of the solution being provided to the customer." – J. Vester*

I am excited for the 2021 DZone Application Security Trend Report because of the relative and important information shared by DZone community members. Readers will explore security solutions for cloud-based mobile and IoT apps, as well as best practices for securing CI/CD pipelines — which is always appreciated for those employing declarative integration and deployment patterns. What would an AppSec report be without a focus on authentication and identity management? Short answer, there is no need to worry — you'll dive into these important aspects of application architecture and design, too.

Readers will also discover a much-needed application security checklist, which everyone who is participating in the application or service lifecycle should bookmark. Referring to my conversation with the feature developer, this checklist will become part of the information provided to all team members on my projects.

Have a really great day!



John Vester

John Vester, Technical Architect at CleanSlate Technology Group

Information Technology professional with 30+ years expertise in application design and architecture, feature development, project management, system administration and team supervision. Currently focusing on enterprise architecture/application design utilizing object-oriented programming languages and frameworks.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2021 Application Security Survey

By John Esposito, PhD, Technical Architect at 6st Technologies

In August 2021, DZone surveyed software developers, architects, and other IT professionals in order to understand the state of application security.

Major research targets were:

1. Security-aimed development and testing techniques used well and poorly
2. Where responsibility for application security is located
3. The role of application security in the SDLC

Methods:

We created a survey and distributed it to a global audience of software professionals. Question formats included multiple choice, free response, and ranking. Survey links were distributed via email to an opt-in subscriber list and popups on DZone.com. The survey was opened on July 28th and closed on August 13th. The survey recorded 515 responses.

In this report, we review some of our key research findings. Many secondary findings of interest are not included here. Additional findings will be published piecemeal on DZone.com.

Research Target One: Security-Aimed Development and Testing Techniques

Motivations:

1. No software is secure unless the creator's thought and effort make it so. This is difficult both in practice (the enemy can always be more clever and/or more resource-endowed than you) and in principle (every expression has a Gödel number). We wanted to know how software professionals are both applying their own intelligence and standing on earlier giants' shoulders to keep their applications secure.

2. Security is a concern at many levels — some are not an application developer's concern (your cleanest Java won't block Spectre-level attacks), but many are, such as:

    - The full entailments of authentication and authorization schemata are hard and often counterintuitive to imagine.
    - The impedance mismatch between the algebra of elliptic fields and the stepwise execution of a Turing machine is a nontrivial gulf to bridge.
    - Any application implementing a protocol can be flooded if the possible requester pool is large enough, now quite easily and cheaply by DDoS over networks of dumb "smart" devices.

    We wanted to understand how much of the security burden application developers take on and are given (and whether these match).

3. Writing code is creation. The enemy is nonsense and chaos. Security is war. The enemy is another mind possibly far more clever than you.

    The creative, productive developer mindset, with a purely logical problem in mind, does not always map well onto the defensive, destructive mindset appropriate for security. While some software professionals (the security specialists)

live perpetually in the war zone, most application developers spend much of their head-time in the build zone. We wanted to see how well the militant mentality required for security mingles with the builder mentality required for application development.

4. In our own experience (largely in enterprise software consulting), most software is terrifyingly insecure. Modern code's house of cards seems ready to collapse for many reasons, not least the shockingly low level of concern for security lingering in applications that perhaps once were never connected to the Internet but now, for some reason, inevitably (if sometimes indirectly) are.

   We also frequently encounter "security-washing" — when people throw a static code scan here, a port scanner there, and maybe even a dependency version check; they call this perfunctory effort to "security" and consider their hands clean (while desperately hoping that nobody tries to venture off the happy path they wrote in their application specification). We wanted to see how widely "security-washing" and "cargo-cult security" (imitation of security's appearance without its substance) have spread, and how aware developers are of these phenomena when they appear.

USE OF APPLICATION SECURITY ARCHITECTURAL PATTERNS

Application architecture involves transformations between business logic and computational logic domains. The resulting bidirectional interplay allows some design patterns to emerge "for free." For instance, the same resource is often needed in many different parts of a transaction (the singleton pattern), and the same productive activity often involves some variations that the caller need not care about (the factory pattern).

These particular object-oriented design patterns are often invented as natural solutions to the business<–>computational mapping without any conscious reference to some magical GoF book. But neither of these domains necessarily say anything about attackers, so security must be injected into application design exogenously. Thinking with secure application architecture patterns is one way to do this.

We wanted to know how often software professionals encounter certain common secure application architecture patterns. Because all patterns are risks for surface-level, insubstantial imitation ("cargo-culting"), we also wanted to distinguish between good and bad implementations of these patterns. So we asked:

How often have you seen good implementations of each of the following application security architectural patterns?

and

How often have you seen bad implementations of each of the following application security architectural patterns?

Answer choices were provided with definitions as follows:

- **Single Access Point** – All access attempts must pass through a single chunk of code (e.g., login screen, unified identity provider) before any access is allowed.
- **Check Point** – Authentication and authorization plus repercussions for failure (e.g., lockout after n password attempts, user access level demotion or data encryption after post-login penetration attempts detected).
- **Security Roles** – Grouping of permissions into buckets and assignments of users to those buckets (rather than assigning each user's permissions individually).
- **Explicit Sessions** – Sandboxing global variables within a single entry/exit sequence (at any level from a POSIX thread to a higher-level session construct like a UDP socket) (e.g., making current user profile information available as a single variable to all code within a session).
- **Error Messaging Over Hiding** – Expose all functionality to all users, but log and/or display errors when user attempts something not permitted.
- **Hiding Over Error Messaging** – Do not allow users to see any functionality they cannot perform.
- **Secure Access Layer** – Encapsulate any lower-level security (e.g., data encryption) within your application's data access layer (i.e., do not require/expect each service/method/class to handle its own encryption).

Results for good implementations encountered (n=470):

Figure 1

***GOOD* IMPLEMENTATIONS OF APPLICATION SECURITY ARCHITECTURAL PATTERNS**

	Single access point	Check point	Security roles	Explicit sessions	Error messaging over hiding	Hiding over error messaging	Secure access layer
Often	62.0%	45.8%	57.3%	29.4%	30.8%	45.5%	50.6%
Sometimes	27.9%	37.3%	32.5%	32.0%	32.5%	31.7%	32.4%
Rarely	7.0%	13.3%	8.1%	27.9%	24.4%	15.3%	11.8%
Never	3.0%	3.6%	2.1%	10.7%	12.2%	7.4%	5.2%
n=	469	467	468	459	467	470	466

Results for bad implementations encountered (n=467):

Figure 2

***BAD* IMPLEMENTATIONS OF APPLICATION SECURITY ARCHITECTURAL PATTERNS**

	Single access point	Check point	Security roles	Explicit sessions	Error messaging over hiding	Hiding over error messaging	Secure access layer
Often	25.5%	21.3%	24.6%	22.7%	27.0%	22.0%	25.4%
Sometimes	39.1%	43.7%	38.4%	38.1%	38.7%	40.9%	37.0%
Rarely	26.6%	25.2%	27.2%	26.8%	24.8%	26.5%	26.3%
Never	8.8%	9.9%	9.9%	12.4%	9.6%	10.6%	11.3%
n=	466	465	464	467	460	464	460

Observations:

1. The top three application security patterns whose *good* implementations are most likely to be encountered often (single access point, security roles, secure access layer) were also three of the top four most likely to have *bad* implementations encountered often.

   It seems that, while these three "filter"-type security patterns are often implemented, the quality of implementation can be improved significantly.

2. Implementations of blocking prohibited functionality at the interface layer (hiding over mirror messaging) are more likely to be encountered in *good* implementations than in *bad* implementations to any extent (92.6% vs. 89.4%, respectively).

   This finding is interesting because this pattern seems especially easily misapplied — for example, when some prohibited functionality is not rendered on a client machine but is not re-checked on the server. Perhaps this is such an obvious mistake that application developers are especially conscious of it and therefore less likely to implement functionality hiding poorly.

3. Error messaging over hiding is the security pattern most likely to be implemented poorly often (27%).

   Since error messaging interacts with failure handling in general, which is also implemented poorly, we suspect that some nontrivial portion of this pattern's implementations that respondents considered bad may not necessarily result in security holes but rather simply in bad UX and difficult debugging. For example, showing a "this operation is not permitted" message is surely not a *good* implementation of this pattern, but it also does not allow unauthorized access.

4. Secure access layer is the pattern second least likely to be encountered in *bad* implementations at all — i.e., is most likely to have never been encountered in *bad* implementations (11.3%). Further, this pattern is far more likely to be encountered often in a *good* implementation (50.6%) vs. sometimes (32.4%) and again vs. rarely (11.8%).

   This is consistent with our anecdotal observation that the dictum, "don't roll your own crypto" (an instance of this pattern), is widespread — and with the fact that maintaining a separate data access layer is one of the simpler, more obvious, and cleanly separable patterns implemented in any layered application architecture, security-conscious or not. So there is less pressure from other architecture-level considerations to implement secure access layers poorly than to implement, for example, explicit session management, which does not automatically play nicely with stateless paradigms.

5. Explicit sessions is the pattern least likely to be encountered in *bad* implementations at all — i.e., is most likely to have never been encountered in *bad* implementations (12.4%).

   But it also shows one of the two most middling *good* implementation encounter likeliness curves. That is, *good* implementations of explicit sessions are likely to be encountered often, sometimes, or never in roughly equal numbers. This may result from the fairly tight interaction of explicit sessions with other aspects of software architecture (discussed in the previous observation), resulting in less independence of explicit session implementations from the non-security-focused aspects of the code.

6. Senior respondents (>5 years' experience as a software professional) are significantly more likely to encounter *good* implementations often — on average 10% more — than junior respondents, and they are somewhat less likely to encounter *bad* implementations often — on average 5% less.

   So more experience raises respondents' evaluation of the quality of application security at the architecture level. This is a good sign: It suggests that software architecture is more secure than if the numbers had been flipped.

USE OF SECURE CODING TECHNIQUES

Much security must be maintained at a lower conceptual level than architecture. Buffer overflows, perhaps the most famous and intuitive exploit type, are as low-level as supra-OS programming gets — a matter of memory (or namespace or another fixed, finite set) management. Also, while it is possible to atomize software design and task assignment such that many individual developers need not think much about architecture, no developer, in a language that requires explicit memory allocation, can afford not to worry about buffer overflows. Nor can developers whose code involves any I/O safely avoid thoughts about input validation or sanitization.

We wanted to get closer to the code execution level and investigate secure coding techniques, so we asked:

How often do you use the following secure coding techniques? Rank from most commonly used (top) to least commonly used (bottom).

Results (n=402):

(See Table 1 on the next page)

Table 1

FREQUENCY USING SECURE CODING TECHNIQUES			
Technique	**Rank**	**Score**	**n=**
Input validation	1	4,648	402
Secure coding standards	2	3,714	393
Data sanitization: input	3	3,626	384
Deliberate architecture and design sessions for security	4	3,532	372
Principle of lowest possible privilege	5	3,382	367
Whitelisting (permission explicit, denial default)	6	3,225	371
Static code analysis for security	7	3,040	382
Simple design to shrink attack surface	8	2,943	349
Data sanitization: output	9	2,599	345
Penetration testing	10	2,454	359
Policy of ignoring no compiler warnings	11	2,348	332
Threat modeling	12	2,093	320
Defense in depth	13	2,052	321
Fuzz testing	14	1,438	300

Observations:

1. Input validation is easily the most commonly used secure coding technique. It requires less devious/black-hat thinking than the nearest competitor (data sanitization: input) and is often applied without special concern for security (e.g., email address validation is not necessarily a matter of security). This result is not surprising.

2. More interesting is the second highest ranked practice: secure coding standards. The high rank suggests that, at management level, security is considered at least enough to formulate rules.

   Whether those specific rules actually produce secure software and are implemented well is, of course, a separate question. And some later results from our survey (regarding the need to consider security as more important than it currently is) suggest that a significant amount of "security-washing" may be contaminating the use of secure coding practices.

   Further, in principle, although rules can catch many foolish mistakes, they aren't adequate for security in the way that they may be for robustness, for instance, because when solving security problems, the enemy is a clever mind (not something non-intelligent like "network partition"). It is possibly more clever than whoever formulated the rules.

3. Use of secure coding practices varies with experience level. Differences in rank were significant between senior and junior respondents:

   Table 2

DIFFERENCES IN RANKING SECURE CODING TECHNIQUES	
Senior respondents	**Junior respondents**
Input validation	Input validation
Secure coding standards	Deliberate architecture and design sessions for security
Data sanitization: input	Whitelisting (permission explicit, denial default)

(Table continues on next page)

Deliberate architecture and design sessions for security	Data sanitization: input
Principle of lowest possible privilege	Simple design to shrink attack surface
Whitelisting (permission explicit, denial default)	Secure coding standards
Static code analysis for security	Policy of ignoring no compiler warnings
Simple design to shrink attack surface	Principle of lowest possible privilege
Data sanitization: output	Data sanitization: output
Penetration testing	Static code analysis for security
Policy of ignoring no compiler warnings	Penetration testing
Threat modeling	Defense in depth
Defense in depth	Threat modeling
Fuzz testing	Fuzz testing

Junior respondents seem to give more deliberate design-level thought to security (higher rank of deliberate architecture and design sessions for security) and rely more on better general coding practices (higher rank of whitelisting, simple design, and policy of ignoring no compiler warnings) to achieve application security. That is, while all these practices can increase application security, the techniques that apply specifically to app security — secure coding standards, principle of lowest possible privilege, data sanitization — are more likely to be ranked higher (more often) by more senior respondents.

We can imagine many possible causes: junior developers facing more restrictions on their general coding practices, conscious attention paid to security increasing over time, and less trust in junior respondents to handle security (and more offloading onto specialists). We intend to distinguish among these causes in future research.

MESSAGE INTEGRITY VERIFICATION IN APPLICATION CODE

Ensuring the identity of the sent and the received is essential to security, but it is also as fundamental to communication and information theory as it gets. At a high level, this is what Claude Shannon was aiming at when he formalized the relation between information and entropy while trying to prevent solar radiation from making it hard to understand people talking over the phone:

- Error-correcting codes, in the form of Hamming codes, may be the first (or only?) information-theoretic work many developers do.
- Checksums are a trivially convenient way to somewhat ensure that the data received is the data sent.
- Hash chains lie behind source control and cryptocurrency, and they (often arranged into Merkle trees) serve a similar role in both applications.

Sometimes message integrity can be determined at the sub-application level. This is roughly the idea that underlies all hashing. But integrity can easily blur into safety, as when checksums are felt to provide some kind of security. And sometimes the infrastructure cannot know enough to determine message integrity after this blur. For instance, a checksum may prove that the file you downloaded from Joe's website is indeed the file that Joe listed with a checksum on his website. But the checksum does not prove that Joe's file won't crash your computer, encrypt your hard drive for ransom, or just output the wrong character every time a solar flare slaps Earth in just the right way.

We wanted to know how often developers verify message integrity and how often they judged the application level (as distinguished from any infrastructure level) to be the ideal place to do this, so we asked:

How often do you write application code to verify message integrity?

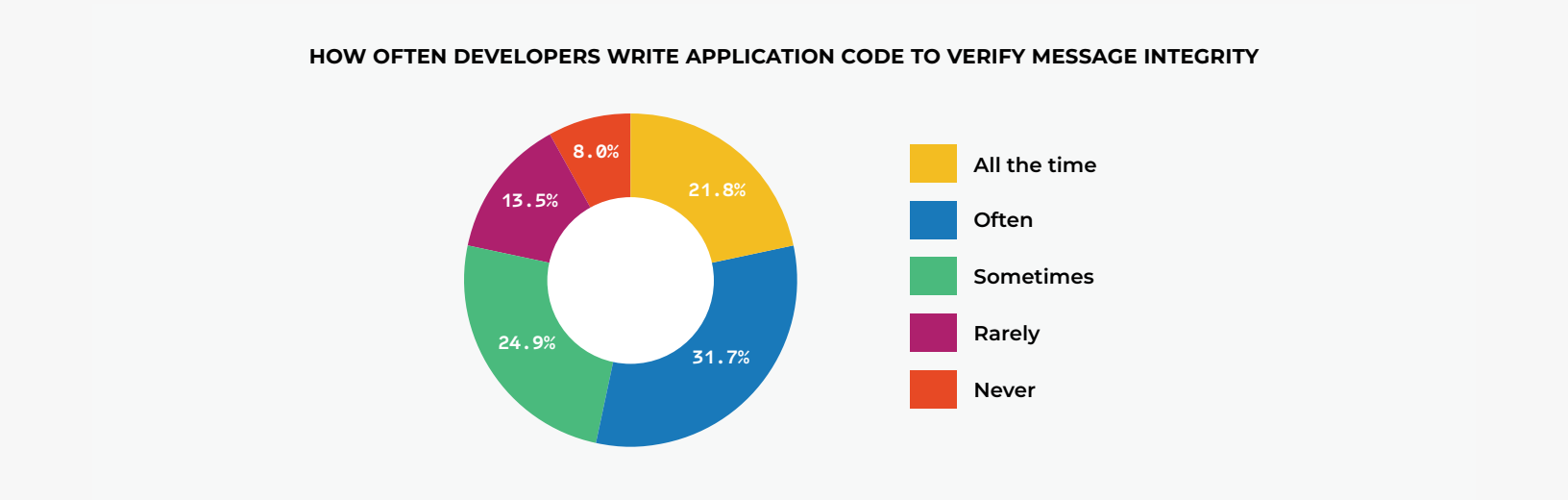
and

(Continues on next page)

Over all the times you've verified message integrity in application code, how often was application code the ideal place to verify message integrity (e.g., because integrity is domain-specific)?

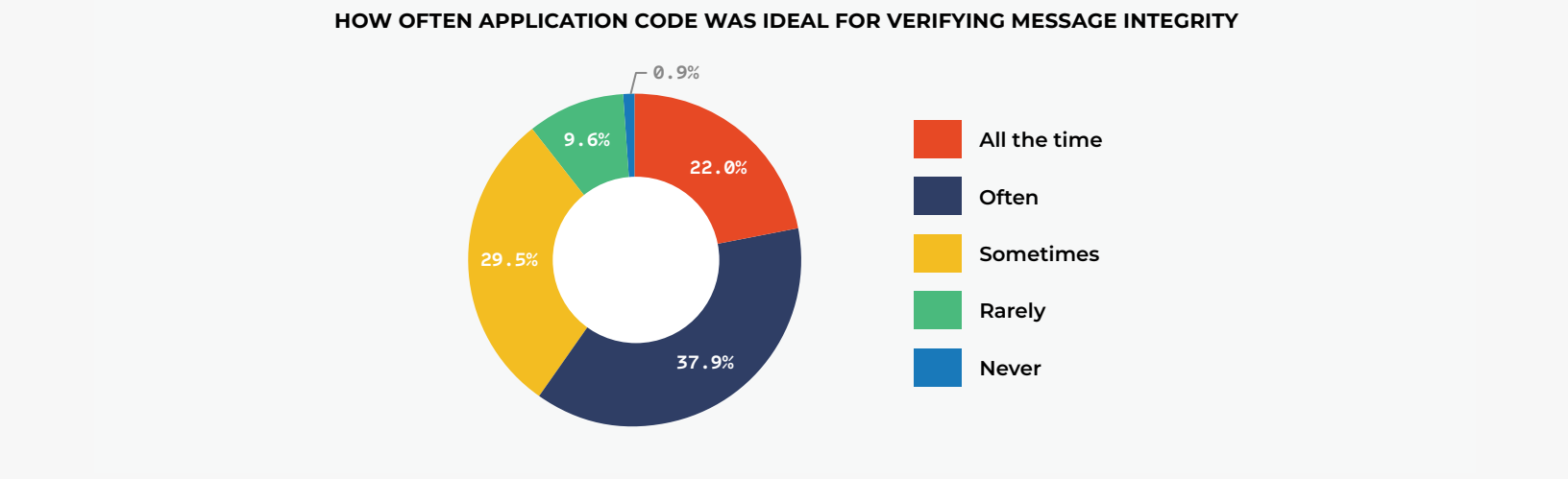
Results for the former (n=473):

Figure 3



Results for the latter (n=427):

Figure 4



Observations:

1. Message integrity is currently treated as a major application-level problem. A third of respondents (31.7%) reported writing application code to verify message integrity often. More than a fifth (21.8%) reported doing so all the time, and another quarter (24.9%) reported sometimes.

   However, there may be room to offload some of this work onto infrastructure. 9.6% of respondents judged that, of all the times they verified message integrity at the application level, it was rarely the best place to do so. Only 22% reported all the time, and almost a third (29.5%) reported that the application level was the ideal place to verify message integrity only sometimes.

   These findings together validate the concern, which we have observed and expressed anecdotally, that too much information-level work is currently left to application developers.

2. Senior respondents are significantly more likely to write application code to verify message integrity (22.7% vs. 15.4% all the time, 33.5% vs. 23.5% sometimes).

    This is consistent with both greater responsibility for security being allocated to senior respondents (see discussion below) and with (our conjecture) senior respondents being tasked with potentially deeper, more theoretically "thick" problems.

ATTITUDES TOWARD FORMAL VERIFICATION AND FUNCTIONAL PROGRAMMING

In theory, formal verification should make a codebase more secure insofar as (a) the set of execution paths through a formally verified program is understood, and (b) the higher-level abstraction offered by formal verification makes a program easier for humans to reason about, thereby avoiding the blind spots that result in vulnerabilities.

Similarly, in theory, functional programming should result in more secure code because (in theory) functional programs are easier to reason about, insofar as side effects are avoided, mappings are explicit, and execution paths can be represented as chained functions — or as nested lambdas or whatever. We wanted to know if both deductively generated claims hold true in the actual experience of software professionals.

FORMAL VERIFICATION AND SECURITY

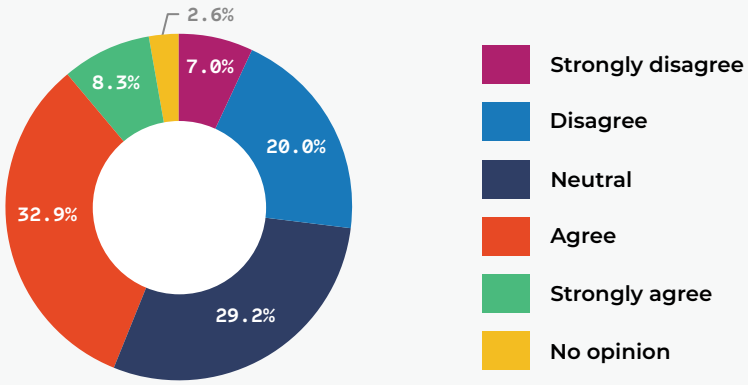
We asked:

Formal verification can solve most application security problems.

Results (n=456):

Figure 5

WHETHER FORMAL VERIFICATION CAN SOLVE MOST APPLICATION SECURITY PROBLEMS



Observation:

Respondents trust formal verification more than we expected. A third (32.9%) affirmed that formal verification can solve most application security problems, another third (29.2%) are neutral, and 8.3% strongly agree. Only 7% strongly disagree with this (rather ambitious) claim. If these results are not an artifact of a poorly worded question or diverse definitions of the scope of app security, then assuming that a very small portion of existing application code is formally verified (on which we have no actual data) — and that respondents are not so inexperienced in formal methods that they overestimate their value — this result suggests that perhaps more time should be invested in formal verification of app code for security reasons.

FUNCTIONAL PROGRAMMING AND SECURITY

We asked:

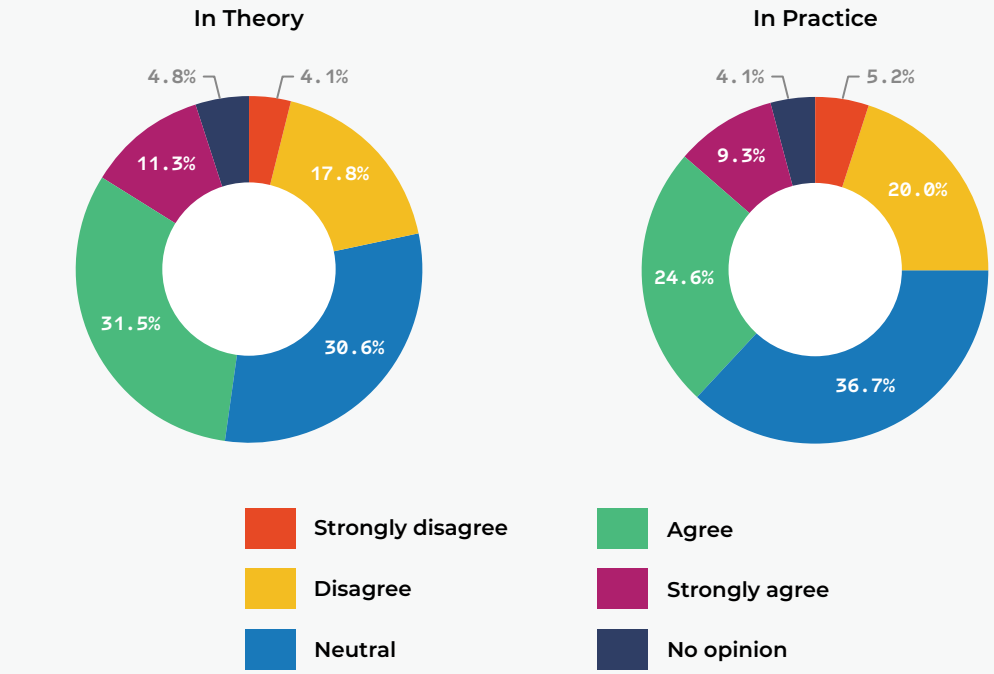
In theory, functional programming results in more secure code.

and

In practice, functional programming results in more secure code.

Results (n=461 and n=460, respectively):

Figure 6



Observation:

There is significant fall-off from *in theory* to *in practice* security benefits from functional programming. 31.5% of respondents agreed that functional programming should, *in theory*, result in more secure code vs. only 24.6% of respondents agreeing *in practice*. 11.3% strongly agreed with this claim *in theory* vs. 9.3% *in practice*, with a corresponding increase in "disagree" responses from *in theory* to *in practice*. These results are not surprising: It would be more surprising if functional programming resulted in more secure code *in practice* than *in theory*.

LITMUS ATTACK SCENARIO: SQL INJECTION

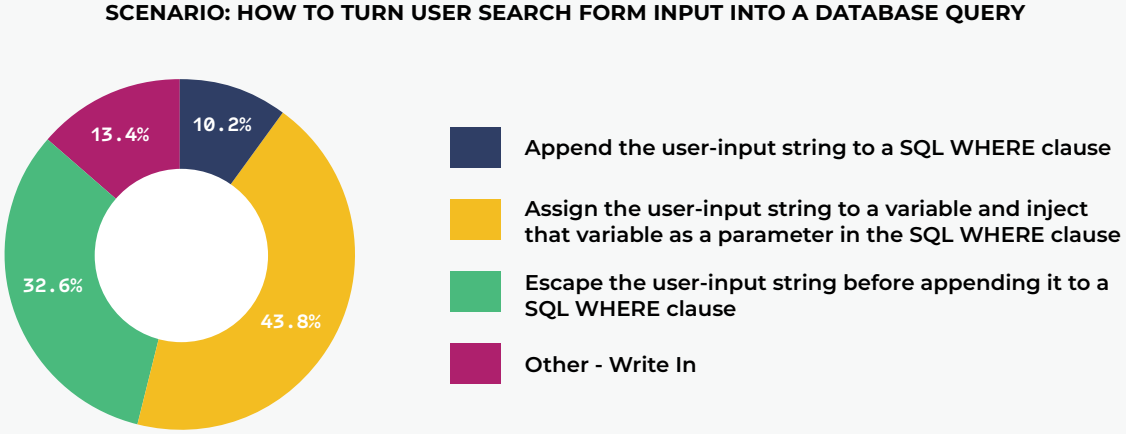
Of the many kinds of malicious inputs, SQL injection is one of the easiest to understand and exploit. Presumably, this is because SQL is relatively easy to read, and many CRUD applications involve dynamically generating SQL statements (since a well-built DBMS is optimized for fetch/update work). As a litmus test of our respondents, especially when distinguishing segments of respondents, we presented a scenario to see how they would defend against a SQL injection attack:

On a public webpage, there is a search form that accepts user input. What is the best way to turn the user's search input into a database query?

Responses (n=491):

(See Figure 7 on next page)

Figure 7



Observations:

1. 10.2% of respondents would be vulnerable to SQL injection. This is a painfully high number for such a straightforward vector but is consistent with static-code-analysis-based research and our anecdotal experience in enterprise consulting.

2. An unusually large percent (13.4%) of respondents offered a write-in "other" response. Most involved protection at a level we might consider "below" the application (e.g., prepared statements, parameterized stored procedures, row-level security checks at user level, use of ORM, use of specialized search appliance with a more security-first query language than SQL).

   Many of these approaches are sound and are often more secure than the application-level options presented in the question. But not all application developers can create stored procedures, the use of an ORM is a decision with implications beyond security, and API-first/service-oriented/microservice architecture tends to frown on allowing such lower-level changes to "leak in" from the application layer. In such cases, protection against SQL injection must remain within the application.

3. Senior respondents fare far better on this question than junior respondents. Only 6.7% (n=22) of senior respondents would simply append the user-input string to a SQL `WHERE` clause vs. 16.3% of junior respondents (n=16).

   In future research, we will ask similar questions about other less obvious scenarios and observe whether our litmus tests suggest that senior developers do tend to avoid simple security mistakes at this level.

   Note: The results from this question are consistent with the differences in rankings of common secure coding between senior and junior respondents discussed below — senior respondents tend to rely more on specific secure coding techniques, while junior respondents tend to rely more on good coding techniques that also result in more secure code.

Research Target Two: Where Responsibility for Application Security is Located

Motivations:

1. The line between writing secure code and securing written code is not easily drawn, and the problem of secure application code cannot be solved by blithe catch-all "shift left."

   As noted above, the builder mindset is not necessarily the defender mindset, and as such, the "enemy" of a programmer is something like formlessness and certainly not malicious intelligence. Not knowing exactly where this line ought best be drawn, we wanted to understand where software professionals currently draw it.

2. Application security may be a side effect of good coding practices, a checkbox to tick for a regulatory requirement, a marketing claim to make a product more sellable, a reaction to an earlier breach, a response to recommendations from

security awareness organizations, and so forth. We wanted to understand where consideration for application security comes from.

3. Because solving application security problems involves both highly rigorous and somewhat free-form ("adversarial") thinking, it seems that the responsibility would tend to fall on more experienced software professionals who have won more hard-fought battles in unforeseen territory. We wanted to see if this turns out to be true.

PERSONAL RESPONSIBILITY FOR APPLICATION SECURITY

Every developer is responsible for making their code do what it is supposed to. But it is not obvious from the structure of a user story, or from developer compensation structures, whether and how much developers are also responsible for their code doing things it is not supposed to. We wanted to know how much software professionals feel personally responsible for the security of any application they work on and how well this individual sense aligns with their employers' expectations, so we asked:

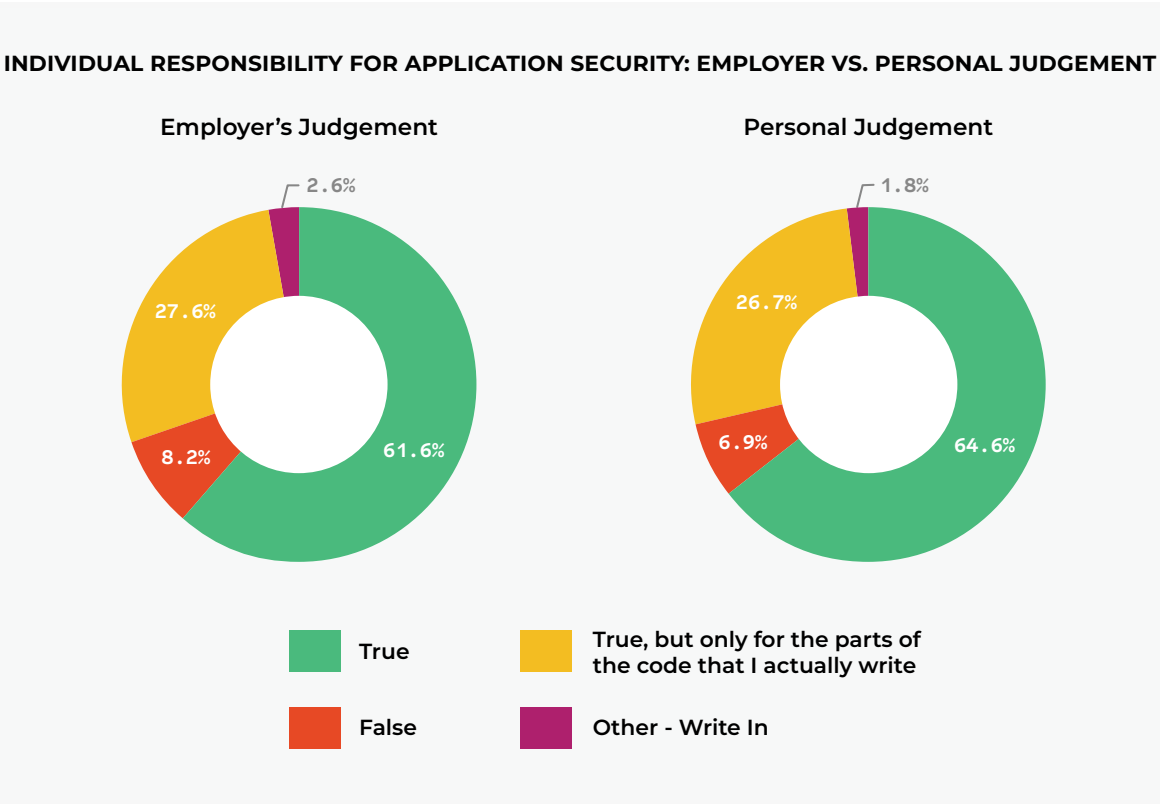
In my employer's judgment, I am personally responsible for the security of any application I work on.

and

In my own judgment, I am personally responsible for the security of any application I work on.

Results (n=497 and n=494, respectively):

Figure 8



Observations:

1. Respondents hold themselves more personally responsible for application security than their employers do (64.6% vs. 61.6%).

   On the assumption that responsibility for things employers don't care about comes from craftsperson pride, aesthetics, or a general sense of responsibility to the users, this result suggests that developers don't perceive app security as something imposed exogenously but rather a part of what they see themselves doing professionally.

2. More than twice as many respondents see themselves as responsible for the security of the application that they work on *simpliciter* as see themselves responsible for the security of the app code that they actually work on (64.6% vs. 26.7%).

    This indicates a strong sense of shared responsibility for security. In future research, we will expand this sense of shared responsibility to cover other areas of software quality (as in the extreme programming and sequent paradigms), and we will be interested to see if respondents feel especially high shared responsibility for security, in particular.

3. In respondents' judgment, technical architects and their employers consider technical architects responsible for overall application security at similar rates (77.4% and 75%, respectively). But developers consider themselves significantly more responsible for overall security than their employers do (56.4% vs. 51.3%).

    Insofar as architects and developers both evaluate their security skills correctly, this result suggests that employers may beneficially shift some security responsibilities to developers rather than architects — perhaps those security responsibilities that involve specific coding techniques rather than architectural patterns, as discussed above.

EXOGENOUS INFLUENCES ON APPLICATION SECURITY DECISIONS

Presumably, if someone has previously stolen your users' personal information, you will take extra precautions to prevent future intrusions. Or if your industry is subject to specific regulations, such as PCI compliance, you will of course follow them or go out of business. But it is not always clear how much external considerations influence security, and in any kind of militant activity, it is far from obvious that foreign impositions suffice for optimal defense (whence e.g., *Auftragstaktik*).

We wanted to know how much exogenous forces have on security decisions in practice, so we asked:

How much impact do the following have on your application security decisions? Rank from greatest impact (top) to least impact (bottom).

Results (n=296):

Table 3

INFLUENCERS OF APPLICATION SECURITY DECISIONS

Technique	Rank	Score	n=
Regulatory requirements	1	2,390	292
Security awareness organizations (OWASP, SANS, etc.)	2	2,292	296
Customer requirements	3	1,965	268
An actual breach at your organization	4	1,635	243
Security makes our software more marketable	5	1,599	261
Actual breaches at other organizations	6	1,589	254
Users tell us about vulnerabilities in our software	7	1,234	238
Demands from executives	8	1,149	228
Demands from investors	9	964	213
Other	10	294	129

Observations:

1. In the judgment of respondents, among external influences, security-focused institutional behavior has the most impact on security decisions by a considerable margin.

    This does not indicate whether these institutions are doing a good job, or whether in their absence software would be more or less secure. But it does suggest the danger of security-washing, in which security is added to software solely or mainly in order to please a non-malicious actor (the state or a professional organization).

Because it's possible to, for instance, follow the OWASP top 10 without thinking like an adversary, it seems that the top two influencers of security decisions are more likely to produce software that avoid foolish mistakes susceptible to published CVEs but not necessarily exert pressure to defend against innovative zero-day attacks.

2. Surprisingly, marketability of security and actual breaches at other organizations ranked at almost the same level.

   Since we did not specify what sort of security decisions these factors influenced, we speculate optimistically that the coincidence of scores is not meaningful. We imagine that marketing affects different security decisions, which are likely to be rather general because they must be recognizable to potential customers, than actual breaches, which are likely to be rather specific because a particular path was exploited. A pessimistic speculation would be that short-term profits influence security decisions more than awareness of actual dangers. In future research, we will select between these options by mapping security influencers to specific security decisions.

Research Target Three: The Role of Application Security in the SDLC

Motivations:

1. Insofar as the builder and defender mindsets do not automatically meld, it is especially important to orchestrate those two approaches over the course of the SDLC. We wanted to know when and how software professionals think about security from requirements gathering to penetration ("pen") testing.
2. Security, like any other software desideratum, has direct costs to both software (performance, architectural elegance, code readability) and users (degraded experience, hours burned), as well as indirect costs (opportunity cost, wider distribution of knowledge of the system). We wanted to understand how software professionals think about application security in relation to other desiderata.
3. As feature sets grow, the attack surface grows; as complexity grows, the attack surface grows. Complexity often grows superlinearly in relation to features. In proportion, feature introduction may result in superlinear growth of the attack surface. We wanted to understand how vulnerable software professionals feel to security issues on each release.

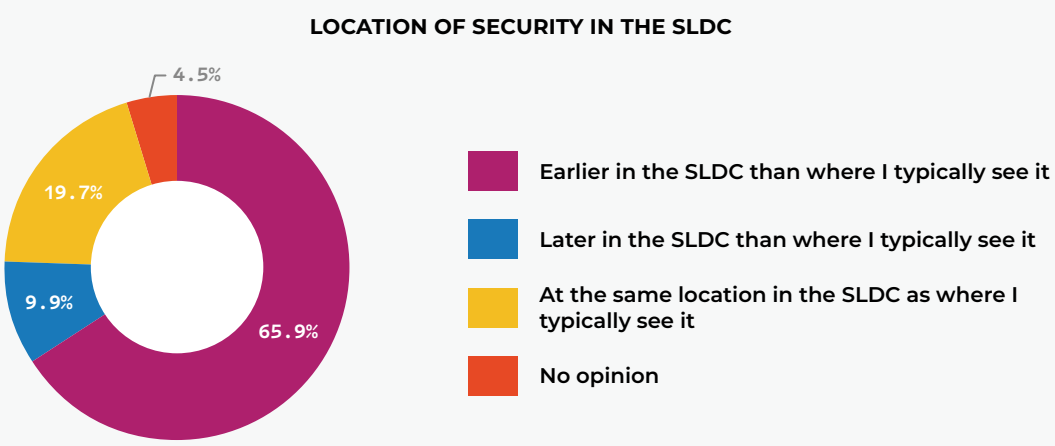
LOCATION OF SECURITY IN THE SDLC

Anecdata, hand-wringing articles from security professionals and marketing blitz from AppSec solution providers, all suggest that more consideration should be given to security earlier in the SDLC. We wanted to see if software professionals in general agree, so we asked:

Security should explicitly be considered: {Earlier in the SDLC than where I typically see it, Later in the SDLC than where I typically see it, At the same location in the SDLC as where I typically see it, No opinion}

Results (n=463):

Figure 9



Observations:

1. Nearly two thirds (65.9%) of respondents agreed that security should "shift left" in the SDLC. Less than one percent of respondents are security professionals. The strong preference among respondents for more security considerations earlier in the SDLC *a fortiori* supports the security specialists' allied claim.

2. The vast majority of respondents (95.5%) have some opinion about the location of security considerations in the SDLC.

   This suggests more interest in security than apathy among developers. In conjunction with the greater self-imposed personal responsibility vs. responsibility employers impose on them, we conjecture developers are actually more interested in security than the state of application code might suggest.

DESIRED CHANGES IN SECURITY-FOCUSED DEVELOPMENT PRACTICES

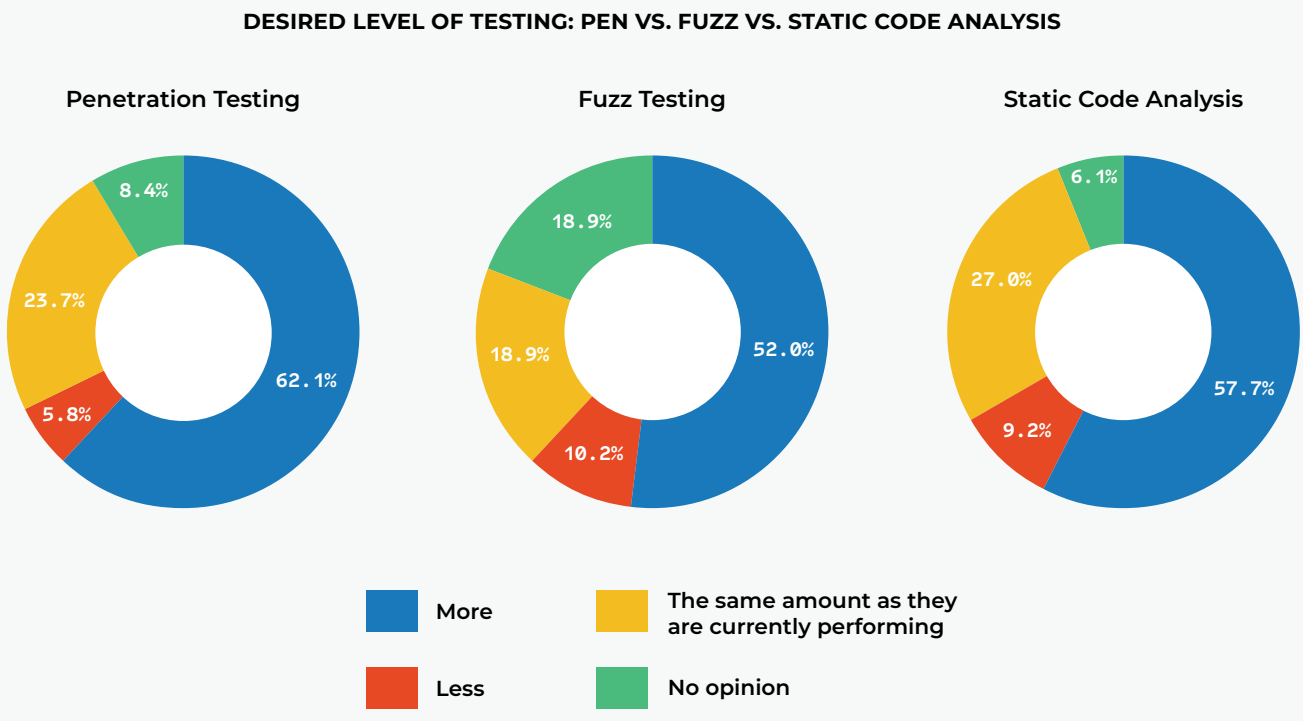
Software creators often see many solutions to a single problem and thereby accumulate many roads not traveled. As a result, they often have a sense of what should be done more or less than currently is being done. We wanted to know whether any such thoughts applied to a few high-level, somewhat automatable secure development practices, so we asked:

My organization should perform:

- *More penetration testing / Less penetrating testing / The same amount of penetrating testing as they are currently performing / No opinion*
- *More fuzz testing / Less fuzz testing / The same amount of fuzz testing as they are currently performing / No opinion*
- *More static code analysis for security / Less static code analysis for security / The same amount of static code analysis for security as they are currently performing / No opinion*

Results (n=464, n=460, and n=450, respectively):

Figure 10



Observations:

1. A majority of respondents want more pen testing, fuzz testing, and static code analysis for security.
2. Of these three activities, pen testing received the most "want more" votes (62.1% vs. 52% and 57.75%). This is the most adversarial minded of the three, which suggests developers think that a more adversarial mindset would benefit application security significantly.
3. Static code analysis for security was the activity most likely to be judged as being performed in the correct measure (27% vs. 23.7% and 18.9%).

    We interpret this as an indication of greater maturity in static code analysis for security, possibly because static code analysis for general code quality is already widespread and adding security-focused static code analysis might require merely appending a few rules to the expression-parsing set. The fact that 78.7% of respondents reported already using static code analysis for application security (in reply to a separate question) testing is consistent with interpretation.

CONFIDENCE IN SECURITY OF RELEASED CODE

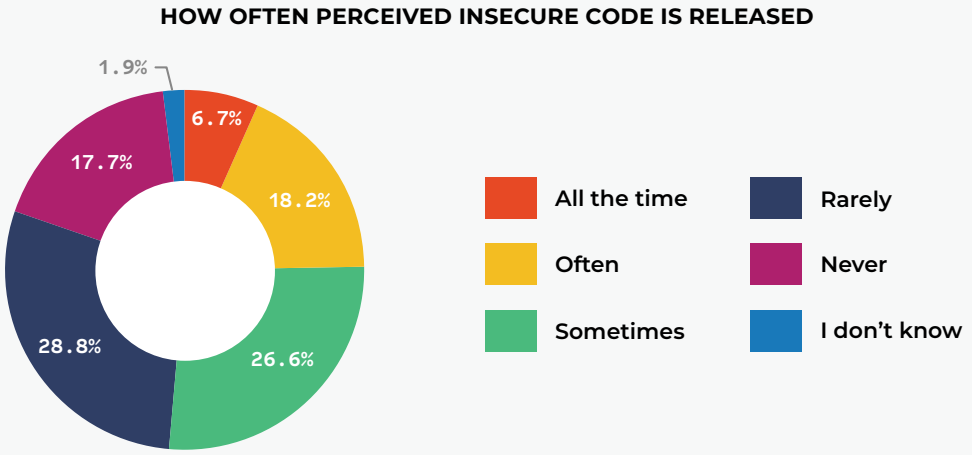
We assume that software professionals' intuitions about the state of released code are not entirely worthless and are especially important to keep in mind when objective answers are impossible to gather (as in the case of never-exploited, exploited but not publicized, or not-yet-but-will-be-exploited vulnerabilities).

So we asked:

How often do you release code that you are not confident is secure?

Responses (n=462):

Figure 11



Observation:

Almost a fifth (18.2%) of respondents reported often releasing code that they are not confident is secure, and 6.7% reported that they do so all the time. To us, these are frighteningly high but unsurprising numbers. Because respondents are aware that the code released may not be secure, they may be able to make it more secure given sufficient resources.

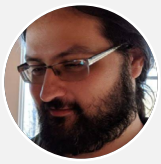
However, we plan to distinguish this possibility — where some ability to make code more secure lies behind the lack of confidence in the released code's security — from simple lack of confidence, which might arise, for example, from awareness of insufficient testing.

Future Research

This analysis only scratches the surface of the topics covered in our survey, our survey only scratched the surface of application security, and AppSec only scratches the surface of software security in general. By the transitive property of metaphorical surface-scratching, we expect that our research into cybersecurity will expand many times in depth and scope.

Topics not covered in this report, but treated in our survey, include:

- Security tools and tests in DevOps pipelines
- Time spent securing application code
- Ranking of security relative to other software development goals
- Presence of security specialists in respondents' organizations
- Techniques used to detect intrusion attempts
- Frequency of excess trust in clients/callers



John Esposito, PhD, Technical Architect at 6st Technologies

@subwayprophet on GitHub | @johnesposito on DZone

John Esposito works as technical architect at 6st Technologies, teaches undergrads whenever they will listen, and moonlights as research analyst at DZone.com. He wrote his first C in junior high and is finally starting to understand JavaScript NaN%. When he isn't annoyed at code written by his past self, John hangs out with his wife and cats Gilgamesh and Behemoth, who look and act like their names.

You don't need the cloud.

Store data with unparalleled security on the decentralized cloud. Get default encryption, multi-layer access controls, and global data redundancy for securely building and scaling apps, streaming video, backing up data and more.

Start for Free →

STORJ | DCS

storj.io

CONTRIBUTOR INSIGHTS

Securing Cloud-Native Applications

Addressing Challenges, Automating Pipelines, and Shifting Security Left

By Samir Behara, Platform Architect at EBSCO

Organizations are rapidly embracing cloud-native design patterns to modernize their business operations and enable faster time to market. Cloud-native architecture combines technologies like microservices, containers, automated CI/CD pipelines, container orchestration, unified observability, and cloud infrastructure. However, modern-day cloud computing services face security risks like data breaches, application vulnerabilities, account hijacking, insecure APIs, malicious insiders, data loss, denial of service, and insufficient credential management.

To protect against these threats, organizations should adopt a zero-trust model for their data and services, as well as embrace the DevSecOps movement to integrate security practices throughout their software development lifecycle (SDLC). Enterprises are using container technologies like Docker to simplify the packaging and deployment workflow of their cloud-native applications. Kubernetes is a popular container orchestration system for automating containerized application deployment, scaling, and management.

Adopting DevOps principles enables teams to deploy and release features rapidly, but it also introduces challenges to the security of your cloud-native applications. Hence, introducing security practices into DevOps is required.

This article will:

- Explain the four Cs of cloud-native security
- Overview cloud-native security challenges
- Showcase security considerations for cloud-native applications
- Explain how to integrate security into your CI/CD pipeline

Four Cs of Cloud-Native Security

The four Cs of cloud-native security are cloud, clusters, containers, and code. Each layer, as seen in Figure 1 on the next page, benefits from the outer security layers, with the code layer built on top of the cloud, cluster, and container layers. Let's briefly go through the recommendations at each layer to secure your architecture:

- **Cloud** – It is the foundation of all security layers, and every cloud provider (like AWS, Microsoft Azure, Google Cloud, IBM Cloud) provides infrastructure security for the running workloads.
- **Clusters** – The primary security concerns in this layer are RBAC authorization, secrets management, pod security policies, and network policies, and Kubernetes is the standard operating tool.
- **Containers** – The security postures recommended in this layer are container vulnerability scanning, image signing, and prohibiting privileged users.
- **Code** – Organizations have the most control over this layer and can implement security recommendations like performing static code analyses, adopting DevSecOps practices, and making security part of the CI/CD pipeline.

Figure 1: The four Cs of cloud-native security

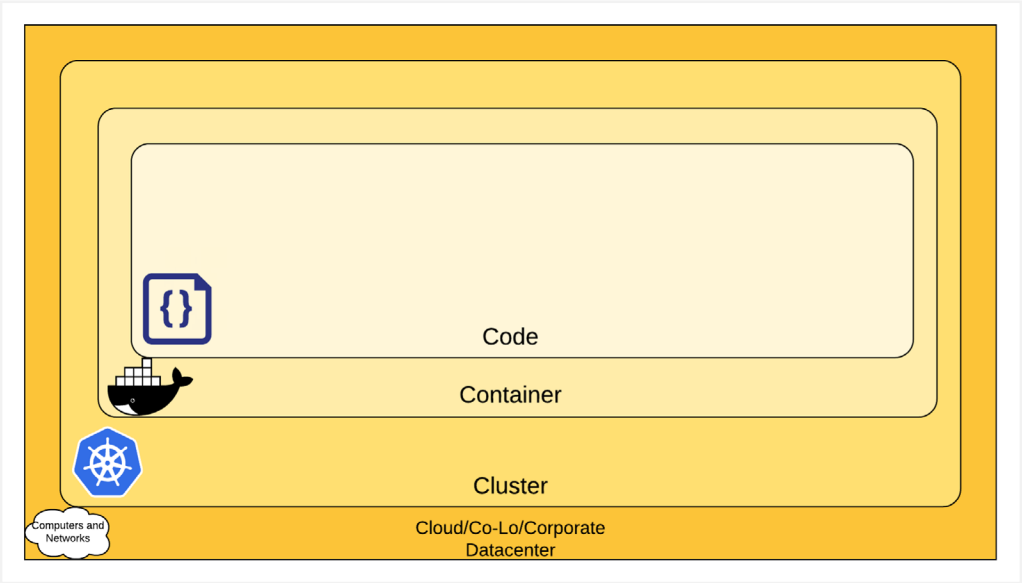


Image Source: "Overview of Cloud Native Security," Kubernetes Documentation

Cloud-Native Security Challenges

With organizations embracing a remote work model due to the COVID-19 pandemic, the need to secure data and systems has become more critical than ever before. The "Zero Trust" security model can improve your organization's security posture in general, especially if they plan to adopt a mobile workforce or hybrid workspace environment. The basic principle is to avoid blindly trusting everything within the security parameter (corporate network) and to always authenticate and authorize every user, application, and device that tries to access the network — whether internal or external to the organization.

The key technologies and principles to build zero-trust-based architectures are:

- Multi-factor authentication
- Identity and access management
- Visibility into device inventory
- Firewall management
- Data classification and encryption
- Least privilege access
- Continuous network traffic monitoring

Cloud-Native Application Security

Containers make it easy to package and deploy the runtime dependencies of your application and help resolve configuration management issues between your development and production environments. However, containers are transient and generally have a short life span, making container security challenging compared to traditional security mechanisms for threat detection and vulnerability scanning.

CONTAINER SECURITY

Out of the box, containers provide some level of isolation and security, but at the same time, they also introduce a set of security concerns like kernel exploits, denial of service attacks, poisoned images, container breakouts, and compromised secrets. Reducing the container attack surface is critical since issues in one container can potentially impact other instances running on the same host. Applying the principle of least privilege and having restrictive user access to containers are good practices. A secret management solution is required to store credentials securely and allow the containers to access sensitive data during operation.

In addition, container images are immutable, so any vulnerability in the container image will persist for the life of the image. Therefore, you need to ensure that the images are regularly scanned for vulnerabilities as part of your CI/CD pipeline, updated with the latest patches, and pulled from trusted registries. When running your cloud-native applications in a containerized environment, you need complete visibility into your cluster configuration and to have monitoring in place.

SHARED RESPONSIBILITY MODEL FOR SECURITY

In the public cloud, security is a shared responsibility between the cloud service provider and its customers. The differentiation of responsibility can be viewed as security "of" the cloud vs. security "in" the cloud. The cloud provider protects the overall infrastructure where the services are running and takes care of the operational concerns in the physical and network layer. On the other hand, customers are responsible for their business logic, including the application code and data layer protection.

Figure 2 illustrates Microsoft's shared responsibility model, including areas of responsibility between the customer and Microsoft for workloads running on-premise and in the cloud.

Figure 2: Microsoft's shared responsibility model

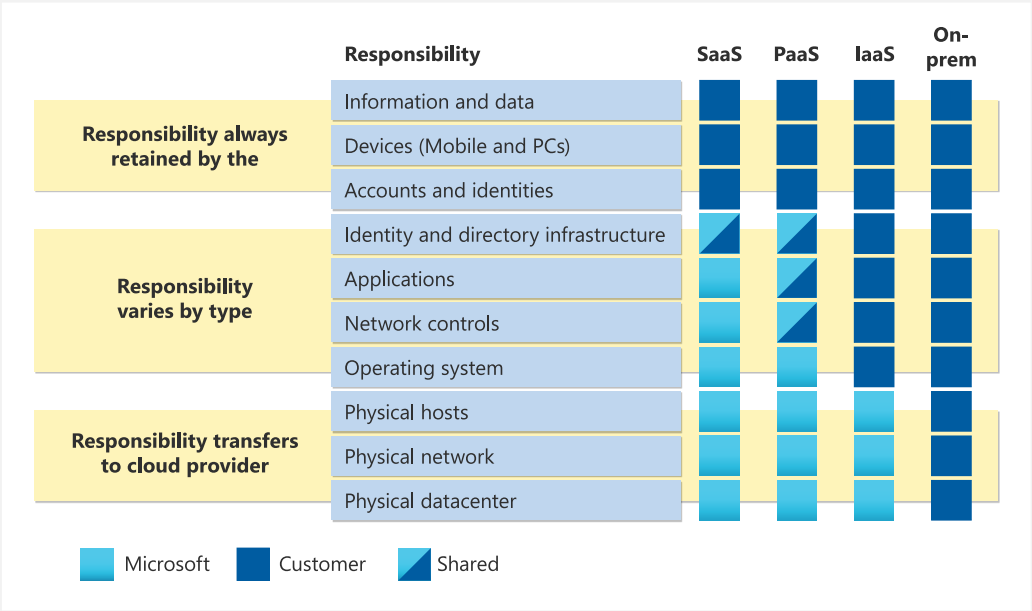


Image Source: "Shared responsibility in the cloud," Microsoft Documentation

Automating Cloud-Native Security

The DevOps methodology focuses on increasing collaboration and transparency between development and operations processes. However, in the pursuit of quick time to market, security practices must not be neglected and pushed further downstream in the pipeline. This is where DevSecOps comes into the picture, incorporating operations and security measures earlier in the development cycle.

SHIFT-LEFT SECURITY STRATEGY

Shifting security left in the development process is essential because you don't want security to be an afterthought. Instead, you should design and build systems with security top of mind. It is expensive and time-consuming to identify and fix security vulnerabilities in production, so the goal of shifting security left is to implement security practices and perform security testing during the development process — and not just before deployment to production. Your DevOps pipeline should allow you to deploy software with quality and speed while also adhering to security best practices.

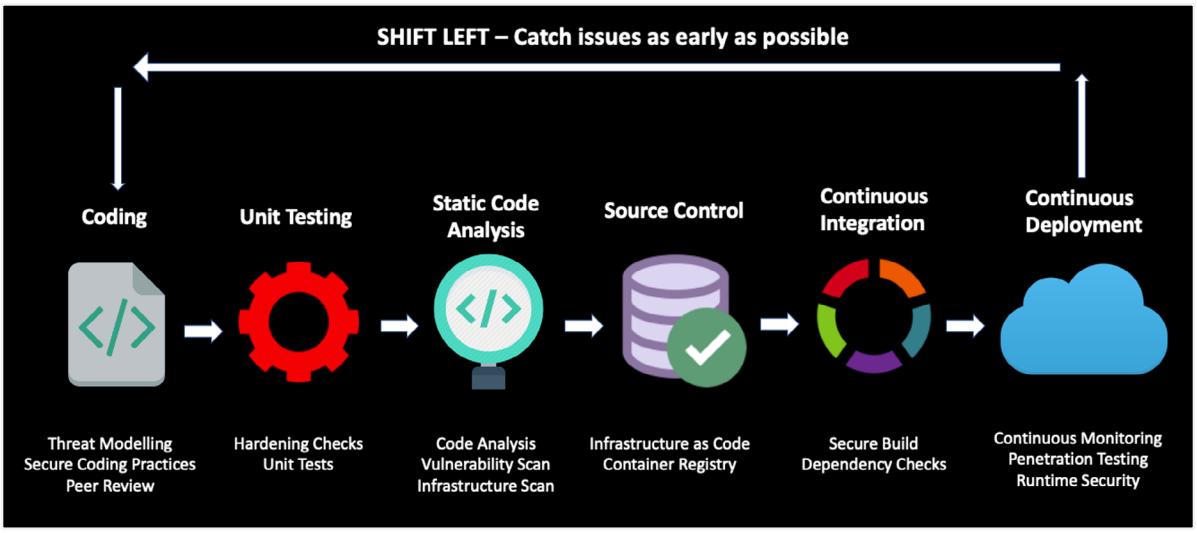
Most vulnerabilities are introduced at the application level, and an attacker can compromise your system if your application contains vulnerabilities. Static application security testing (SAST) and SAST tools enable you to scan your entire codebase and execute security-related rules to detect vulnerabilities like SQL injection, cross-site scripting, denial of service, and code injection issues. In addition, as part of the continuous integration workflow, you can perform static code analysis for every code commit and implement automated vulnerability detection and compliance reporting.

INJECTING SECURITY INTO CI/CD PIPELINES

Integrating security controls into your automated pipelines, as seen in Figure 3, is key to successfully delivering high-quality software. The DevOps pipeline has the required permissions to deploy changes into your environment, so you must have stringent security guardrails around it. Developers can leverage several open-source and commercial security tools for their CI/CD pipelines. The goal is to identify security issues early, and low friction measures like following secure coding practices, the peer review process, and performing static code analysis are easier to implement.

If you are using open-source software or third-party libraries, you should have tooling in place for vulnerability management and threat detection. Implementing a lightweight penetration test in your pipeline helps enhance the security posture — for example, a dynamic application security testing (DAST) tool like OWASP ZAP acts as a security gate before deploying code to production. Along with scanning your containers, consider scanning your infrastructure code before deploying to the cloud.

Figure 3: Building security into the DevOps pipeline



Conclusion

With the cultural shift of building security into DevOps, development teams have an additional responsibility to automate application security testing and integrate it into the release pipeline. Training development teams on security principles and best practices can help bridge the knowledge gap. In addition, development teams working closely in collaboration with IT security teams can help both mitigate security vulnerabilities earlier in the SDLC and achieve the shift-left concept in the context of application development and security.

The Open Web Application Security Project (OWASP) is a non-profit global community that promotes application security mechanisms. It is highly recommended that developers working in the cloud-native space acclimate themselves with the OWASP Top 10 application security vulnerabilities, how hackers exploit them, and methods to eliminate these risks. Most of these vulnerabilities — like code injection, sensitive data exposure, broken authentication, security misconfigurations, cross-site scripting, and insufficient logging/monitoring — can be addressed with secure coding practices and following a stringent code review checklist.

Samir Behara, Platform Architect at EBSCO

@samirbehara on DZone | @samirbehara on Twitter | Author of samirbehara.com

Samir Behara is a Platform Architect with EBSCO and builds software solutions using cutting edge technologies. He is a Microsoft Data Platform MVP with over 15 years of IT experience. Samir is a frequent speaker at technical conferences and is the Co-Chapter Lead of the Steel City SQL Server user group.

CONTRIBUTOR INSIGHTS

Application Security Checklist

Evaluating Security Controls With OWASP's Top 10 Security Tests

By Sudip Sengupta, Technical Writer at Javelynn

In today's technology landscape, organizations are supported by web applications that act as essential enablers to streamlining operations. While these applications enable automation, wider collaboration, and ease of sharing data, they also act as vectors that are prone to malicious attacks. Besides this, as modern applications rely on loosely connected components and services in constant communication, security becomes a complex, time-consuming challenge.

The **Online Web Application Security Project (OWASP)** Foundation seeks to help organizations develop secure applications by issuing guidelines on available tools, techniques, and documentation. The Application Security Checklist is one of OWASP's repositories that offers guidance to assess, identify, and remediate web security issues. This article delves into various vulnerabilities of web applications and outlines OWASP's guidance on testing to mitigate such vulnerabilities.

Selecting the Right Application Security Tests

OWASP's application security testing checklist is an essential guide to promote repeatable and methodological testing for dynamic apps. The following section delves into the workflow and specific activities involved in web app security testing.

APPLICATION TESTING WORKFLOW

A typical application security testing strategy is based on a collection of several common steps:

- Gathering comprehensive information of the application and its platform to assess related technologies and vulnerabilities
- Exploiting the system to test the severity of discovered vulnerabilities
- Ranking vulnerabilities based on the outcome of exploits and risks
- Using vulnerability risk data to re-assess application security posture
- Successful exploitations to be escalated for required mitigation

(Continued on next page)

Figure 1: A typical OWASP Application Security workflow

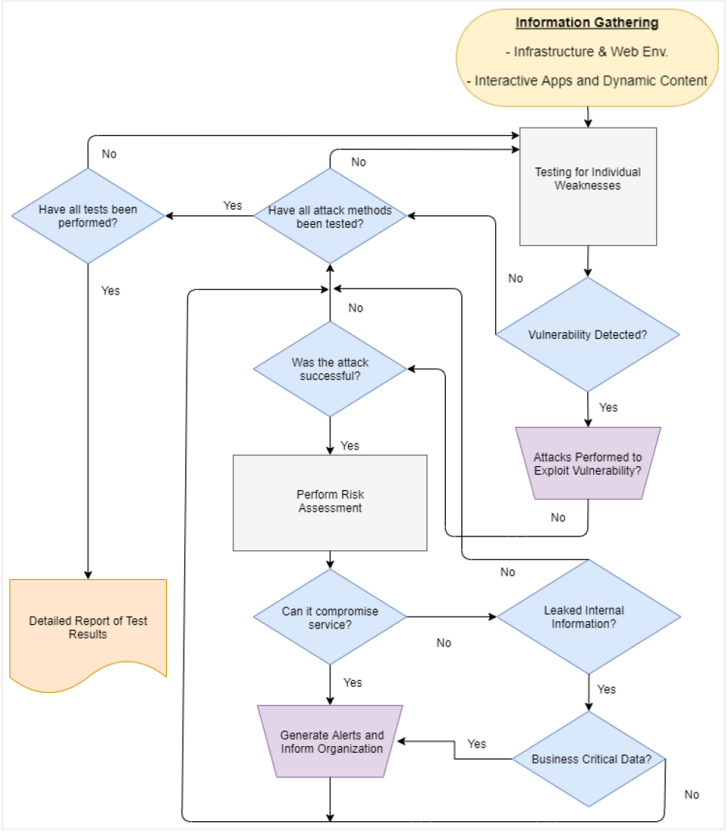


Image source: "OWASP Web Application Penetration Checklist," OWASP

Application Security Testing Checklist

The OWASP Application Security Testing checklist helps achieve an iterative and systematic approach of evaluating existing security controls alongside active analysis of vulnerabilities. Below is a list of key processes and items to be reviewed when verifying the effectiveness of application security controls:

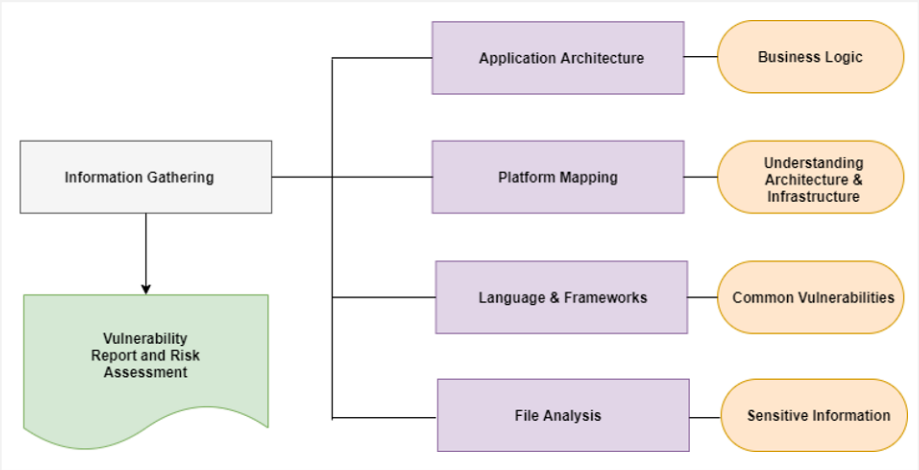
1. INFORMATION GATHERING

A successful web application security strategy fundamentally begins with an understanding of the interactions between the **web server**, **users**, and **applications**. While application deployment platforms vary, key vulnerabilities in infrastructure configuration act as a common weak link for threat actors to initiate an attack.

Some key application security information gathering activities include:

- ☐ Manual site exploration
- ☐ Crawling for hidden content
- ☐ Checking for files that store and expose content
- ☐ Scanning caches on search engines of public sites
- ☐ Web application fingerprinting
- ☐ Identification of user roles
- ☐ Identification of application entry points
- ☐ Identifying related applications
- ☐ Identifying ports and hostnames
- ☐ Identifying third-party hosted content

Figure 2: The OWASP Information Gathering Model



2. CONFIGURATION DEPLOYMENT AND MANAGEMENT

A web server ecosystem is intrinsically complex with highly connected, heterogeneous services and components working together. Reviewing and managing the configuration of the server is, as a result, a very crucial aspect for maintaining robust security across multiple layers of an application.

Securing various configuration items of an application involves:

- ☐ Checking for commonly used URLs
- ☐ Testing network infrastructure configuration
- ☐ Enumerating administrator interfaces
- ☐ Checking supported HTTP methods and Cross-Site Tracing (XST)
- ☐ Reviewing old unreferenced and backup files for sensitive information
- ☐ Testing for Strict-Transport-Security
- ☐ Testing file permissions
- ☐ Testing for non-production data in live environments, and production data in dev/test environments
- ☐ Testing for content security
- ☐ Evaluating subdomain takeover
- ☐ Analyzing client-side code for sensitive data

3. IDENTITY AND ACCESS MANAGEMENT (IAM)

Securing organizational data involves defining appropriate access privileges and roles of the application's users/administrators. Each individual (user, app, or device) gets a single digital identity (also referred as tokens) that can be monitored, maintained, and modified throughout their data access sessions. Assessing the robustness of IAM for application security typically involves testing the following:

- ☐ Role definitions
- ☐ User registration processes
- ☐ Account provisioning processes
- ☐ Account enumeration and *guessable* user accounts
- ☐ Weak or unenforced username policies

4. AUTHENTICATION TESTING

Authentication enforces application security by enabling the web server to verify that a network entity is who they claim to be. As attackers tend to develop unique techniques to bypass authentication schemes, not every authentication method guarantees effective security controls, and requires a continuous assessment process. Assessing authentication security involves the regular testing of:

- ☐ Default credentials
- ☐ Vulnerabilities of the "Remember Password" feature
- ☐ Browser cache vulnerabilities
- ☐ Weak password policies
- ☐ Credentials transported over an unencrypted channel

Testing for sensitive information sent via unencrypted channels involves checking whether credentials are encrypted or encoded, and sent as HTTP headers using a `curl` command of the form:

```
$ curl -kis http://darwin.com/restricted/
HTTP/1.1 401 Authorization Required
Date: Fri, 28 Aug 2021 00:00:00 GMT
WWW-Authenticate: Basic realm="Restricted Area"
Accept-Ranges: bytes Vary:
Accept-Encoding Content-Length: 162
Content-Type: text/html

<html><head><title>401 Authorization Required</title></head>
<body bgcolor=white> <h1>401 Authorization Required</h1>  Invalid login credentials!
</body></html>
```

5. SESSION MANAGEMENT

Once a user is authenticated, their interaction with the server is managed within a **session**. Improperly managed sessions open doors for attackers to compromise access mechanisms by assuming those to be identities of legitimate users. More so, such compromised accesses are often taken advantage of by attack vectors that escalate privileges and penetrate deeper into the system. To avoid vulnerabilities within a session, the following processes are recommended to be tested as a best practice:

- ☐ Analyzing session tokens for cookie flags
- ☐ Checking session cookie durations
- ☐ Examining termination after a relative timeout
- ☐ Testing for the possibility of single-user multiple sessions
- ☐ Testing for consistent session management
- ☐ Testing cookies for randomness

6. CRYPTOGRAPHY

Cryptography ensures the secure exchange of information by using algorithms that transform human-readable data into a **ciphertext-encrypted** output. While doing so, the process establishes trust between the web server and network entities using security keys, making it an important mechanism for maintaining application security. Testing cryptography for maintaining application security involves:

- ☐ Checking for sensitive, unencrypted data
- ☐ Testing for the usage of wrong algorithms
- ☐ Testing algorithm strength
- ☐ Analyzing functions for randomness
- ☐ Checking for the appropriate usage of salting

7. CLIENT-SIDE TESTING

Since full-blown attacks carried out on the perimeter are usually challenged by effective organizational security efforts, threat actors tend to favor smaller, repeated attacks to gain initial access to web servers. To mitigate such approaches, client-side or internal testing involves examining vulnerabilities on applications installed on an endpoint that communicates with the web server. Client-side testing reveals weak points that can be exploited using the access rights of authorized users, and includes testing the following:

- ☐ Cross-Site Scripting (XSS)
- ☐ JavaScript execution
- ☐ Client-side URL redirects
- ☐ Cross-Site Flashing (XSF)
- ☐ Web sockets and web messaging
- ☐ Cross-Site Script Inclusion (XSSI)

8. ERROR HANDLING

OWASP encourages developers to include error handling mechanisms and messages that enable them to fix issues of user access. Improper error handling can expose sensitive information such as database dumps, error codes, and stack traces that can be exploited by attack vectors to gain access.

Testing error handling mechanisms can be done through:

- ☐ Testing server behavior for resource requests that are unavailable
- ☐ Testing HTTP RFC for *breaking ambush requests*
- ☐ Observing server behavior when requested for files/folders that do not exist
- ☐ Identifying the application's data entry points
- ☐ Listing and understanding the services configured to respond with error messages

9. DATA VALIDATION

Any information entering a web server's network edge should be tested and verified to ensure that it is in an acceptable format. Data validation testing includes:

- ☐ Examining *special* files
- ☐ Testing file upload validation mechanisms
- ☐ Testing for rich user content validation
- ☐ Assessing content security policy
- ☐ Evaluating the list of regular expressions

10. BUSINESS LOGIC

Hackers mostly leverage an application's original programmed flow to orchestrate breaches and penetration attacks. As a result, it is recommended to assess the business and application's configuration to identify vulnerabilities in code or business logic that could be used for potential exploits.

Business logic testing includes:

- ☐ Testing for feature misuse
- ☐ Testing for non-repudiation
- ☐ Testing trust relationships
- ☐ Testing data integrity
- ☐ Testing for duty segregation

Conclusion

While administering robust security is of utmost importance, OWASP updates its checklist based on the changing security landscape and mistakes of organizations that caused vulnerabilities. The OWASP Top 10 Application Testing Checklist offers a repository of potential vulnerabilities for developers to help enforce security across all layers of a workflow's pipeline. The project includes multiple resources and activities that aid organizations to ensure web applications and their underlying components don't serve as a gateway for malicious actors. The checklist also helps teams formalize their web application security efforts, while minimizing the scope of risk in case of an attack.

Sudip Sengupta, Technical Writer at Javelynn

@ssengupta3 on DZone | @ssengupta3 on LinkedIn | www.javelynn.com

Sudip Sengupta is a TOGAF Certified Solutions Architect with more than 15 years of experience working for global majors such as CSC, Hewlett Packard Enterprise, and DXC Technology. Sudip now works as a full-time tech writer, focusing on cloud, DevOps, SaaS, and cybersecurity. When not writing or reading, he's likely on the squash court or playing chess.

CONTRIBUTOR INSIGHTS

Designing Secure Authentication and Identity Management

Building Better Security Through MFA, Access Controls, and More

By Joey D'Antoni, Principal Consultant at Denny Cherry and Associates Consulting

Organizations and individuals face an ever-increasing threat from a wide variety of actors. Threats can come from nation states, organized crime gangs, or even determined individuals. These attacks come in the forms of ransomware, which can cripple your business and cause data loss or data exfiltration. Beyond ransomware, more insidious attacks like the SolarWinds supply chain attack can impact a large number of organizations beyond the initial attack victim. A supply chain attack looks for weak links in the business process — in this case, pursuing a network monitoring vendor whose software is widely used *and* inherently needs to run with high privileges.

While there is no single complete security solution, a multi-faceted defense and in-depth solution can provide strong protection against attacks. The first layer of that strategy has always been a strong identity management solution to provide authentication and authorization. In this article, you will learn about the design of modern identity management solutions, including multi-factor authentication, just-in-time and conditional access, and how these solutions can integrate with a wide variety of custom and off-the-shelf applications.

What Is Identity Management?

The history of identity management dates to the 1500s when governments began to consistently issue birth certificates. As computer systems became prominent, usernames and passwords for individual computers became prevalent; however, that solution didn't scale to large distributed systems. Local usernames and passwords evolved into identity federation systems like Microsoft's Active Directory, which allow users to log in to multiple systems in a "circle of trust" and provide centralized management and monitoring of authentications. A centralized system allows for both users, as well as services or applications, to have identities. In modern identity and access management (IAM) solutions, application identity is a major part of the security model.

Legacy systems like Active Directory and other identity management solutions rely on lightweight directory access protocol (LDAP) for data storage and Kerberos as its network authentication protocol. One of the limitations of Kerberos were limitations around its HTTP functionality, which limited the ability to modernize authentication systems using a combination of protocols such as SAML, WS-Federation, and OAuth, all of which aim to move away from the classic username and password and rely on token-based claims.

While the user may still authenticate with a username and password, they are issued a token that has specific information about which resources the requestor has access to. These tokens expire and can be revoked, which provides extra levels of security control that contain additional metadata that allows for richer security methods like conditional access.

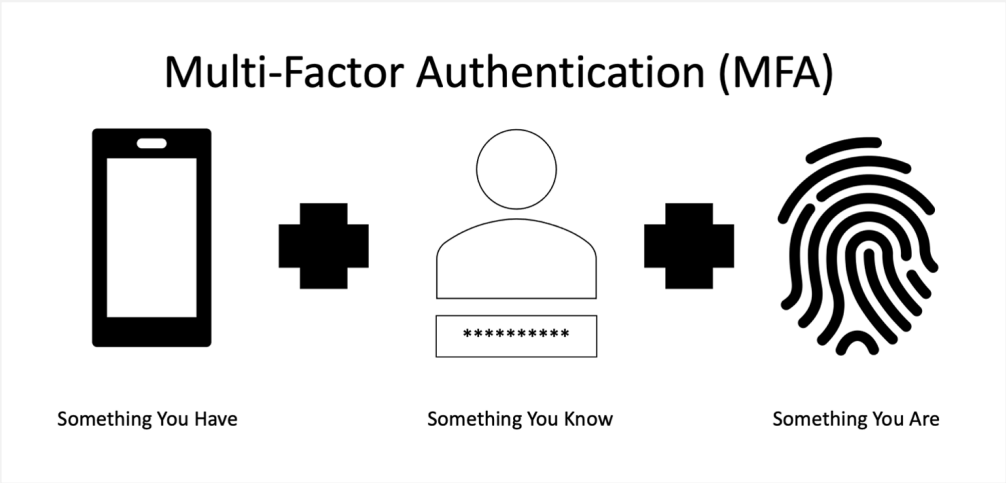
The other aspect of modern IAM systems — like Azure Active Directory and Okta — allows users to authenticate beyond the circle of trust, and to authenticate other software-as-a-service offerings like their own corporate identities. This means that instead of using a specific username and password for each application like Salesforce or DocuSign, users can authenticate with their corporate identity, which allows for better management and security for those applications.

Multi-Factor Authentication — You Need to Do It

Another important aspect of modern security is multi-factor authentication (MFA), which requires the user to have a username and password, and then authenticate using a second device, whether it be in the form of a text message, email, physical key, or an authenticator application on a phone. Authenticator applications are more secure as they are far less likely to be compromised than a cell phone number or an email account.

The image below describes multi-authentication at a high level — modern MFA systems require both a password and PIN or approval from your mobile app, and in some cases, a fingerprint or facial recognition is required as the final step.

Figure 1



Some users may complain about the MFA process as it takes more time than simply logging in; however, the additional security provided is well worth the overhead. The other concern is that your applications support authenticating using MFA, as some legacy applications may not support modern authentication methods. The client access issue is a major concern — in the case of legacy applications, this can be challenging, especially if you lack access to source code to change the client drivers. If you own the source code for your application, or you are building a new application, you should also understand how your authentication stack integrates with your authentication provider.

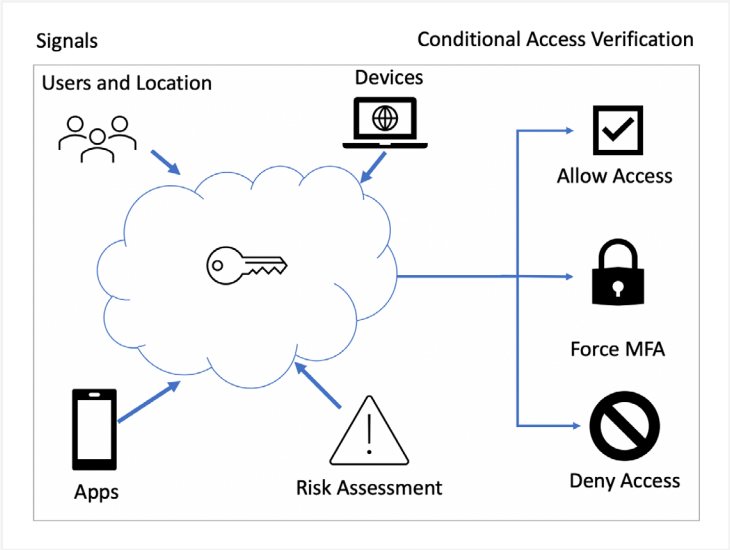
What Is Conditional Access?

Beyond just using MFA, there are other ways you can add additional intelligence into your authentication process. Conditional access uses logic after the initial authentication to decide whether to allow a connection to complete, force an additional MFA process, or block the connection completely. At a basic level, this means if/then logic is implemented when a user authenticates.

Typically, this is implemented through various policies. Some examples of common policies include:

- Requiring MFA for all administrative users
- Blocking access from specific countries
- Limiting access to managed devices for specific applications (for example, email)
- Requiring the device to be currently patched

Figure 2



An example of a how conditional access works is shown in Figure 2.

Signals are defined as the input (or the *if* part of the if/then logic) that conditional access uses for its decision point. There are a number of signals that conditional access platforms use, but some common ones are trusted IP address ranges, what application is making the request, or device state. Using these signals allows conditional access to dynamically change who and what can log in, making your authentication process more secure.

Privileged Identity Management and Just-in-Time Access Controls

The biggest risks to any organization's systems and data is administrative accounts, which can typically perform all manner of attacks, including deleting or exfiltrating data and removing audit logs that demonstrate which activities occurred. In fact, this was the crux of the SolarWinds attack — because of the low level where SolarWinds collected data, its software inherently had to run with extremely high privileges. Attackers are always trying to gather administrative credentials so they can do more damage.

However, organizations need administrators to keep the lights on, so what should they do? A common technique is for administrators to have two sets of credentials — a typical low privileged account for tasks like email and other business systems and a dedicated administrative account for systems management tasks. This has a couple of benefits: It makes auditing admin activity easier since you can narrow the focus to the admin accounts, and secondly, it means admins are not normally logged in with admin privileges, thus reducing the attack footprint.

Privileged identity management and just-in-time access take this split account to another level. The administrative accounts are disabled by default, and the administrator requests access to the account, which is approved by a manager or a fellow administrator. Since access is then limited to a specified amount of time, the scope and processes are widely configurable in most authentication platforms, so you can tailor your policies to meet your organization's needs. The major benefit of PIM and just-in-time access is that administrative logins are disabled by default and a process is put in place to escalate to those accounts.

Conclusion

The modern threat surface can be daunting to many organizations. While implementing multi-factor authentication everywhere possible is a good start in protecting against advanced threat actors, implementing more complex controls like conditional access and privileged identity management are needed to protect your environment from more advanced attacks.

Joey D'Antoni, Principal Consultant at Denny Cherry and Associates Consulting

@jdanton1 on DZone | @jdanton on Twitter | joeydantoni.com

Joseph D'Antoni is a Principal Consultant at Denny Cherry and Associates Consulting. He is recognized as a VMWare vExpert and a Microsoft Data Platform MVP and has over 20 years of experience working in both Fortune 500 and smaller firms. He has worked extensively on database platforms and cloud technologies and has specific expertise in performance tuning, infrastructure, and disaster recovery.

CONTRIBUTOR INSIGHTS

Mobile and IoT Security Strategies in the Cloud

Security Measures and Principles in Azure, Google Cloud, and AWS

Boris Zaikin, Software and Cloud Architect at Nordcloud GmbH

As the Internet of Things (IoT) space continues to expand, "smart" products are becoming more popular. Now you can easily buy and connect devices like vacuum cleaner robots, doorbell cameras, and smart locks. Moreover, you can combine all these devices in a smart home set. However, some product teams don't take security risks into account. And they postpone the introduction of security features into the product until a security attack or data leak occurs.

This article is separated into the following parts:

- Four examples of the most prominent IoT attacks
- Key principles of and best practices for securing IoT applications and devices
- Examples of IoT components and security solutions for Azure, Amazon Web Services (AWS), and Google Cloud (GC)

Most Popular Attacks on Your IoT Environment

1. **Trendnet webcam security breach** – The company released its web cameras without security protection, and anyone could easily access the camera just by knowing the device's IP address. Also, the company stored user login credentials in clear, readable text.
2. **Jeep hack** — Imagine driving your car when suddenly, you lose control and see that someone is driving your car remotely. A few engineers tried to connect to a Jeep through its cellular network and execute an exploit that updated the car's firmware. The result was total control of the car: Hackers managed to slow down and move the car out of the road.
3. **Smart TVs** – Many smart TV sets don't have any authentication options. For example, I started casting YouTube to my neighbor's device by wrongly clicking another TV that appeared in the list of devices on my smartphone. Another example is when hackers can use a TV's microphone and camera to watch what the device owner is doing.
4. **Smartphones** – Hackers can steal sensitive data by running exploits on a mobile phone. In one such case, a user typed something on their smartphone, which created soundwaves that malware recorded, converted to symbols, and sent over the internet.

IoT Security Principles and Best Practices

Now that we understand several ubiquitous security problems in IoT, let's dive into four key security principles and solutions.

WATCH YOUR IOT DEVICES

You should constantly monitor your IoT devices and infrastructure. Device monitoring can be an issue, especially for enterprise companies or industrial factories. For example, if employees bring USB drives that are compromised with malware, you can mitigate it using the following security measures:

- Configure network with firewalls and other security compliance.
- Use central security monitoring that covers all IoT devices in the organization.
- Add specific workstations for user-provided external devices.
- Monitor legacy devices — for example, an old conference smart TV device connected to the central network can be an unsecured backdoor to the whole organization.

USE JSON WEB TOKEN AUTHENTICATION

Use JSON Web Tokens (JWTs) and the latest signature-based standards — for example, JWT ES256 and JWT RS256. JWTs provide one of the safest ways of authentication based on OAuth and OpenID protocol. Here is how JWTs work for IoT:

- The device should provide authentication data (user, password, SSL certificate, unique device ID) for an authentication service that validates this data and generates the JWT.
- The device uses this JWT to access the cloud services and API. The cloud identity services check the token every time before the device tries to access the cloud API or services.

INTEGRATED SECURITY APPROACH

Companies that produce IoT devices should focus on integrating security into device controllers during the production phase because integrating security features in IoT device firmware post-production can be extremely challenging, or nearly impossible in some cases. Securing a completed device may also lead to additional expenses. Often, you need to set up additional security infrastructure, or you must send all devices back to update the firmware — this always means additional time and money. So customers choose devices with security software that is already onboarded.

USE TLS OR LWC EVERYWHERE

IoT device producers should consider hardening their devices to use Transport Layer Security (TLS) or Lightweight Cryptography (LWC). IoT devices should check certificates on the server-side and revoke it if it is compromised. Next, I will focus on using these principles in cases where an organization has IoT infrastructure in the cloud or wants to migrate it to the cloud.

Building Secure IoT Architectures in Azure

In this section, I describe the most prominent Azure resources to build secure IoT architecture, accompanied by an IoT architecture example.

AZURE IOT HUB

Azure IoT Hub is a resource service that allows your solution to communicate with the IoT device — it is a service bus with IoT features that sit like middleware between the device and back-end service of your application. Azure IoT Hub has the following features:

- Registers and stores device data
- Enables device telemetry, data insights, and monitoring
- Supports device-to-cloud communication, request-reply, and file upload from service communication options
- Secures connections based on X.509 certificates and SAS tokens

IOT EDGE

Azure IoT Edge is a platform based on edge computing principles. IoT Edge allows IoT devices to run in offline mode, integrates with Azure IoT Hub, and has modules that run in the IoT device. Each module is a Docker container, and it can be custom code or Azure services-based code. For example, code gathers logs and telemetry or manages the connection between devices and the Azure cloud (Azure IoT Hub). IoT Edge contains the runtime that is also installed on a device and orchestrates the modules.

Azure IoT Edge security functionality includes:

- **Support for confidential computing** – The application or module is encrypted in transit and at rest.
- **TLS-based encrypted certificates** – IoT Edge devices use these between modules, runtime, and the cloud.
- **An additional security layer** – This layer, Security Manager, secures not only modules and runtime software but also the device's hardware layer.

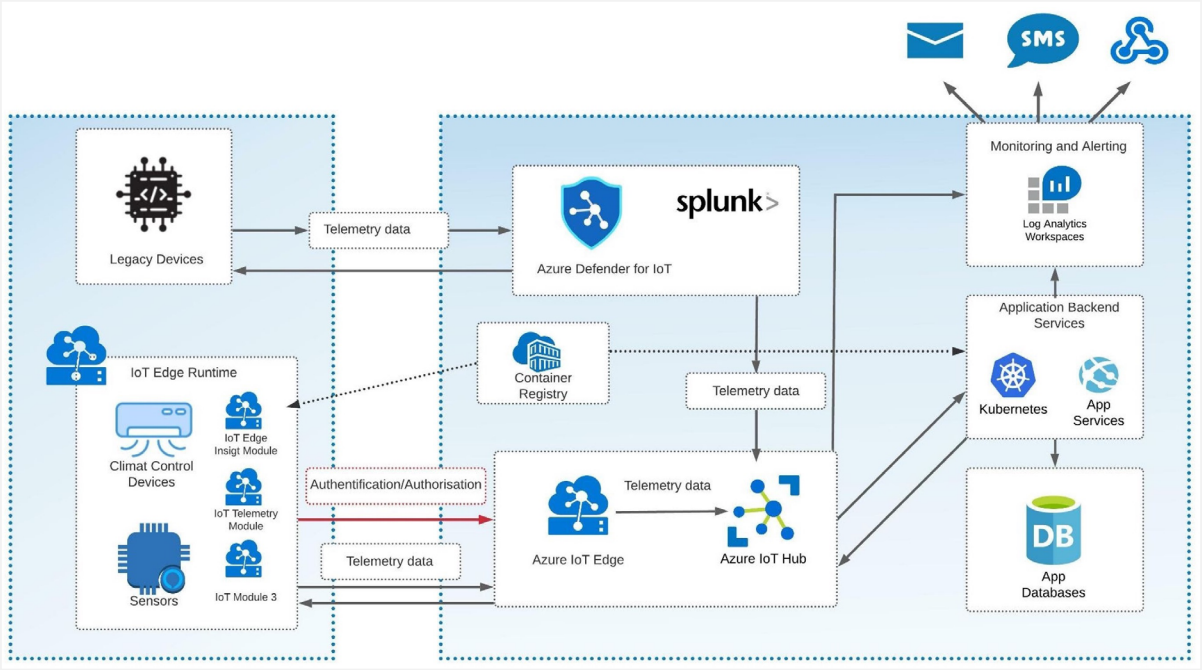
AZURE DEFENDER FOR IOT

Azure Defender for IoT is a security option that allows you to identify security vulnerabilities and threats in your IoT devices and IoT back-end infrastructure. One of its significant advantages is that it supports agentless setup, which is suitable for legacy devices that don't support agent setup. Let's review an example of how your IoT architecture may look using all these components.

ARCHITECTURE EXAMPLE USING AZURE IOT SOLUTIONS

The architecture solution illustrated in Figure 1 below is an intelligent climate control system for warehouses. Climate control should maintain different temperatures, humidity, and air quality according to the season — this architecture is based on a real customer use case.

Figure 1



The IoT device infrastructure for the warehouse climate control system in this example contains: air temperature, humidity, and air quality sensors; climate control devices; and legacy devices that don't support agent setup. Hackers can use legacy devices as a backdoor to the whole system. So we send all data from these devices to Azure Defender for IoT to check for potential vulnerabilities. Also, Azure Defender allows you to reinforce the device infrastructure security by involving tools like Splunk. Splunk integration is part of the Azure Defender for IoT service. Other devices operate via the Azure IoT Edge Runtime directly in the device and in the cloud.

Additional processes that occur:

- Azure IoT Hub stores the device data, and the application back-end services operate with data and manage the climate control system according to the sensor telemetry.
- Back-end services are deployed to Azure Kubernetes Services and App Services.
- Azure Log Analytics builds a monitoring and alerting subsystem that the operator can react to when some security incident happens. Therefore, when a device has security issues or doesn't respond, log analytics display such occurrences in the dashboard and send alert notifications such as email, SMS, and webhook.

Let's see how we can build the same secure architecture in Google Cloud.

Building Secure IoT Architectures in Google Cloud

To build IoT architectures, Google Cloud (GC) provides us with IoT Core, which is a fully managed IoT service that offers the following features:

- Device registration
- Device telemetry aggregation and analysis
- Integrated Pub/Sub
- Device connection management
- Streamlined integration with other data monitoring and processing services for more granular data analysis

IoT Core also provides robust security features:

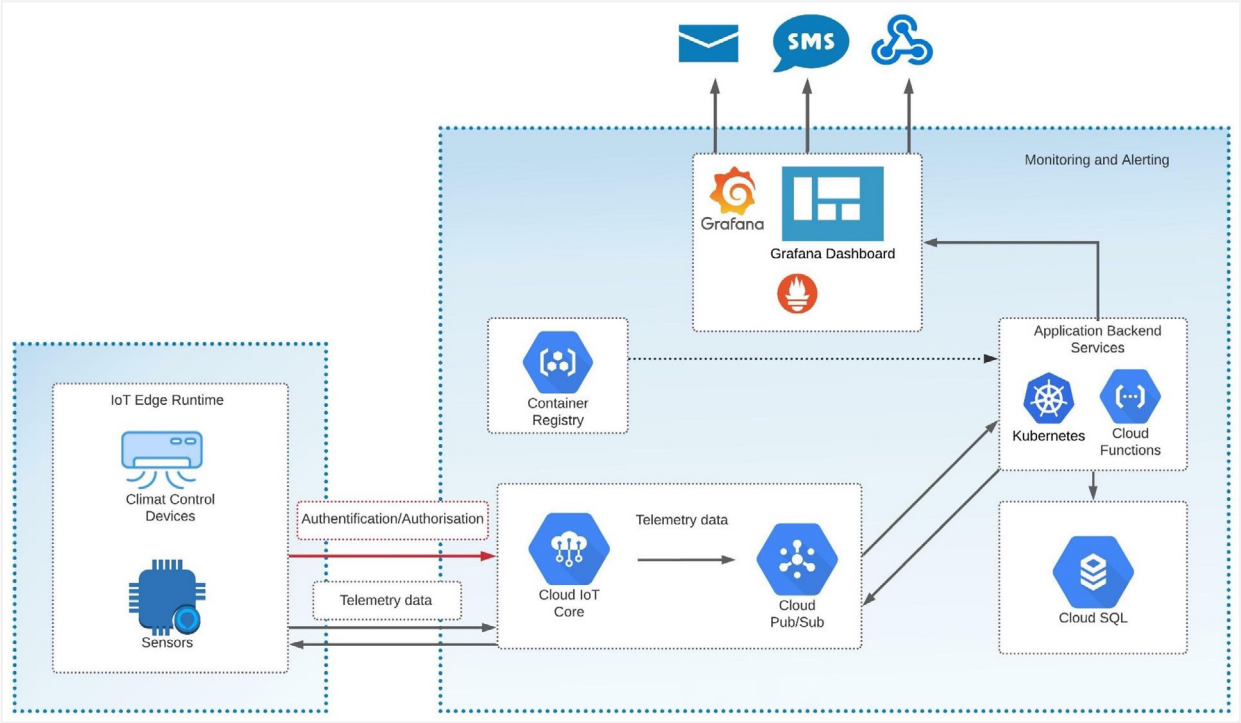
- JSON public and private web tokens with a limited expiration time
- TLS 1.2 certificates
- RSA encryption
- Support for GC Identity and Access Management (IAM) to manage API access

Let's see how the same architecture looks in Google Cloud.

ARCHITECTURE EXAMPLE USING GC IOT CORE

I will keep the same use case — intelligent climate control systems for the warehouses — for this example.

Figure 2



The security workflow in Figure 2 contains an authentication step. The initial authentication process covers the generation of public and private keys. It is the main requirement to register the device in the IoT Core service. Two components are involved in that process: the provisioner and device manager.

With IAM, a user who is assigned a role of provisioner with `cloudiot.provisioner` has permission to manage the devices but cannot modify or delete the registry. The device manager is a component of IoT Core that allows you to register devices and verify device identity. The device manager stores the public key, while the device itself stores the private key. The second phase of authentication is JWT generation. The device generates and signs the token and with a private key, then sends the token to the MQTT bridge, which is a component of IoT Core that verifies the JWT and establishes the connection.

The solution also contains monitoring that uses Prometheus log provider and Grafana dashboards. But if the device can use certificates and JWT tokens, the IoT core can accept it.

Building Secure IoT Architectures in AWS

AWS IoT Core is almost the same as the service in Google Cloud that I described above. It includes the following security functionality:

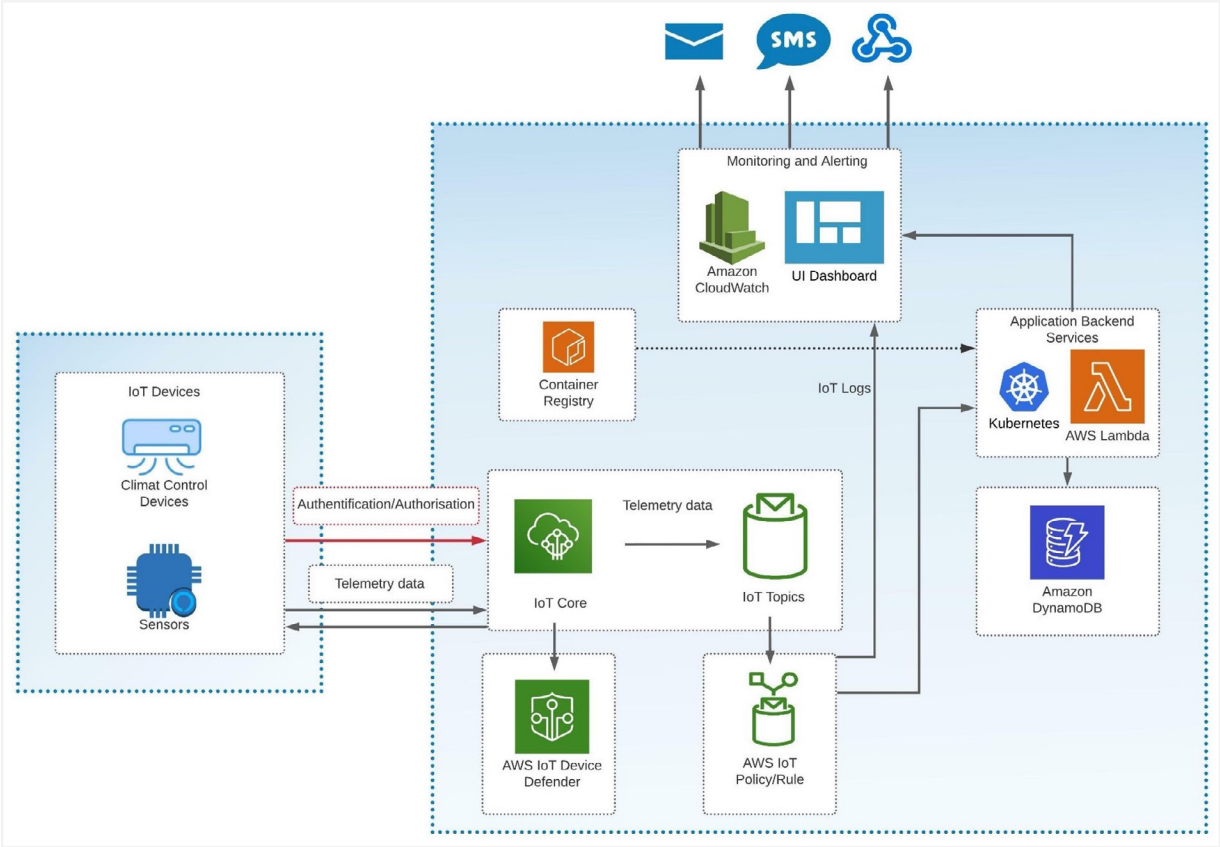
- TLS-encrypted connection (same in the GC IoT Core)
- Fine-grained device permissions based on Thing policy variables

In addition to all security options provided by IoT Core, you can use AWS IoT Device Defender — a service that analyzes device logs and data to find potential security issues. This is similar to the Azure Defender for IoT, covered in the Azure section.

ARCHITECTURE EXAMPLE USING AWS IOT CORE

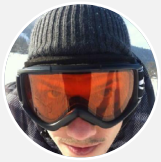
Below, you can see an example of AWS Architecture that is similar to the GC Architecture mentioned above. Everything looks the same except the IoT policy. IoT policy filters data by different topics. Since it can be a category or device ID, your back-end services can listen to a specific topic to collect data. AWS IoT defender is optional here and used as an additional security reinforcement.

Figure 3



Conclusion

In this article, I provided cases of secure IoT solutions, general principles of secure IoT architecture, and example workflows of secure IoT architectures based on Azure services, Google Cloud, and AWS.



Boris Zaikin, Software and Cloud Architect at Nordcloud GmbH

@borisza on DZone | @boris-zaikin on LinkedIn | @boriszn on GitHub

I'm a Certified Software and Cloud Architect who has solid experience designing and developing complex solutions based on the Azure, Google, and AWS clouds. I have expertise in building distributed systems and frameworks based on Kubernetes and Azure Service Fabric. My areas of interest include enterprise cloud solutions, edge computing, high load applications, multitenant distributed systems, IoT solutions.

CONTRIBUTOR INSIGHTS

Accelerate Your CI/CD Pipeline With Security on Autopilot

Transform Your Pipeline Into a Flywheel With Proven Security Practices

By Anita Raj, VP of Product Marketing at ThroughPut Inc.

What Makes a Secure CI/CD Pipeline?

As businesses once again look to grow and scale, many of them are adopting the DevOps and CI/CD approach to ensure continuous value generation and relevance in today's hypercompetitive market. CI/CD certainly enables modern businesses to automate building, testing, and deployment, and thus not just bridge the gap between traditional development and operations teams but also accelerate time to market.

However, this approach also entails an ongoing struggle to maintain quality — not to mention critical security — during frequent releases. Driven by the need to be agile, most DevOps teams end up compromising on security. Thus, CI/CD often generates a challenge for businesses to ensure regulatory compliance, as security policies are mostly introduced much later in the development cycle — dramatically intensifying a whole range of risks.

More importantly, the process of manually verifying compliance with dynamic security policies across entire product cycles, teams, and organizations as they adopt CI/CD at scale is simply ineffective — not to mention impossible.

So how can security automation drive agility in DevOps?

- It drives organizations to implement a "shift-left" strategy that proactively minimizes errors.
- It eliminates belated manual security checks that delay and threaten the build cycle.
- It helps developers to course-correct earlier in the build cycle.
- It assures security teams that the CI/CD pipeline is aligned with the most recent security regulations.
- It ensures continuous, automated compliance without impacting development/operational agility.

Key Challenges in Securing CI/CD Pipelines

Let's face it, the CI/CD pipeline was not conceived with security as a key consideration. Speed, efficiency, and convenience (to read more – https://dzone.com/articles/microservices-a-mere-hype-or-the-future-of-software-development) were the primary goals. There are also reasons for which security has thus far taken a back seat in CI/CD pipelines, one of which is the belief that focusing on security can slow down development.

However, when a threat eventually appears, and it always does, the losses and damage far exceed the advantages gained by the agility of the process anyway. CI/CD pipelines typically consist of several components — such as tools, servers, resources, and containers — that are required to enable fast, efficient development and deployment. These components function at varied levels of configurations and dependencies as they interact with one another, creating an entire ecosystem — one that by sheer virtue of its complexity is impossible to monitor manually, despite being home to several easy-to-miss vulnerabilities.

It is, therefore, important for developers to carefully analyze their CI/CD pipelines and modify them to adopt a security-first approach — after all, a stitch in time does save nine. By merging security into the process of development itself, DevOps (or DevSecOps) teams can ensure a much more robust value offering each time, while also freeing themselves of the constant threat of neglected vulnerabilities.

Best Practices for Integrating Security Into CI/CD Pipelines

Security in the simplest of environments is still hard to get right — and DevOps have famously complex environments. The key to adopting a security-first approach to DevOps is acknowledging the fact that neither is expected to be an expert on security, nor is security expected to be an expert in development. Both need to work around each other to achieve effective outcomes — in this case, high-quality, scalable, and secure applications.

Thus, the best approach is to assimilate security *into* the development process, while ensuring that security remains an enabler — not an obstacle to development. This also means that only essential security checks should be run, such as those in the following table:

Table 1

Security Check	Description
Infrastructure analysis	This is a full scan of the environment, including servers, configurations, etc., to analyze drifts and fix them.
Source code vulnerabilities	The source code is often the target of malicious attacks, so make sure that it has adequate protection to ensure continued application integrity.
OSS library vulnerabilities	Open-source components such as libraries, which are extremely useful when it comes to pushing releases faster, actually increase software vulnerabilities as they obscure visibility and create blind spots, thereby escalating risk.
OSS version	The thing about using open-source software is that it can reach its end of life at any time. When that happens, security is the first thing to take a hit as maintenance stops.
Credential authentication	With multiple people and teams accessing the CI/CD pipeline, authorizing access is a nightmare. However, compromising credentials is one of the most popular means to gain access to proprietary code — which means the only way to prevent access is to ensure error-proof identity authentication.
Static application system testing (SAST)	Also called white-box testing, this is when code vulnerabilities are tested before compiling code. It helps fix issues early on in the lifecycle before they have the opportunity to snowball into larger problems involving complex fixes.
Active and passive penetration testing (dynamic analysis)	This is the testing of application responses to real-time values. It helps identify vulnerabilities once applications are live.

Honestly, it's not as difficult as it looks. There are plenty of tools available in the market to help with these tests — Checkmarx, OWASP Zed Attack Proxy (ZAP), CyberArk, and WhiteSource come to mind.

Pitfalls to Avoid When Scaling Your CI/CD

Scaling quickly while maintaining quality is one of the biggest challenges in DevOps. Since the DevOps culture has traditionally regarded security as an inhibitor when it comes to achieving speed and agility, it goes without saying that a lot of runtime fails are unfortunately tied into security oversights. Some of these include:

- Applications being deployed into production while still housing critical vulnerabilities, and when changes cost more
- Applications being exposed to the internet due to lack of monitored, regulated access
- Compliance issues (since security is often an afterthought and not treated with the same importance as software bugs)
- Lack of zero-trust network security policies that detect and block unexpected communication between components
- Failure in correlating app behavior — sanctioned or unsanctioned

Steps to Implementing Breach-Proof CI/CD Pipelines

DevOps can be truly successful only when it adopts a shift-left approach that embraces and merges security throughout the CI/CD pipeline, thus significantly minimizing the chances of releases that contain exploitable vulnerabilities. In other words, DevOps cannot afford to relegate security testing to the end of the development lifecycle. The best bet is to adopt a defensive stance — and prevent the merest possibility of an exploit. Here are a bunch of practices that can stand you in good stead:

- **Know your pipeline and components** – It is critical to have visibility into the entire CI/CD pipeline, including all its assets, steps, tools, and boundaries, to gain a full understanding of the application architecture and its development lifecycle. You can then start imbibing security into the development and deployment processes. The choice of security solutions may depend on the pipeline components, but many vendors offer solutions that cover most scenarios.
- **Check open-source vulnerabilities** – Analyze all imported open-source resources for known vulnerabilities. These third-party components play a vital role in DevOps, but the vulnerabilities they may introduce can impact the security of an application even if its code is unchanged.
- **Tighten and continuously monitor access control** – Establish access control rules to limit all access to the CI/CD pipeline. Track, monitor, and manage every member's access to every component and resource along the entire length of the pipeline — with task-based, role-based, or even time-based controls. Once controls have been set up, regularly take stock of all assets, verify permissions and configurations, and modify them as tasks and roles evolve.
- **Perform a threat modeling exercise and secure all points of access** – Identify all potential security threats and analyze all connections to the pipeline. Every connection is potentially vulnerable to compromise, so regularly scan and patch all devices and channels that link to the pipeline. All infrastructure configurations, too, should be locked down in place and regularly scanned for vulnerabilities.
- **Set up checks to commit code and quickly analyze committed code** – Ensure all code meets baseline standards for quality as well as security before it is committed. Establish rules and controls to inspect code before committing it into the central repository. Also ensure that developers receive prompt feedback once they commit their code. This is best done with the help of static code analysis tools, as they don't require the application to be running to analyze the code.
- **Meticulously monitor and clean up** – A CI/CD pipeline is a continuous flow of moving parts and processes. However, the nonstop cadence should not lull you into procrastinating routine security maintenance chores. Always monitor the pipeline, and once tasks are completed, terminate any temporary resources and remove any tools and utilities that are no longer required. This reduces the surface area available for malicious attacks.

DevSecOps: No Longer an Oxymoron

Securing your CI/CD pipeline end to end will enable you to release high-quality software at a brisk speed, minus the recalls and PR nightmares of unstable releases. However, this is not possible without close, continuous collaboration across development, operations, and security teams. The idea is to adopt security as a core part of development, not as a checkbox to be ticked post development. Only then can you foster a culture of responsibility for your software in its entirety — and not just for its functionality as a means to an end.

Anita Raj, VP of Product Marketing at ThroughPut Inc.

@anita-raj on DZone | @anita4tech on Twitter | anitars.com

Anita is a results-oriented product leader and marketer who combines her technologist bent of mind with her keen eye for user-centric outcomes to deliver real business outcomes. Currently the VP of Product Marketing at ThroughPut Inc., Anita is focused on setting the organization up for hyper-growth. She is also a member of the prestigious Forbes Communication Council and has been published across multiple platforms including CloudTweaks, Thrive Global, DZone, and PMMHive. In her previous stints, she has worked with multi-million enterprises and high-growth startups across the U.S, UK, and Germany, including EMC, Infosys, Progress, Brillio, LeanIX, Signavio, and more.

CONTRIBUTOR INSIGHTS

Application Security Predictions

Exploring the Potential of Artificial Intelligence and Blockchain Over the Next 12 Months

David Yakobovitch, Host of HumAIn Podcast

Cybersecurity is evolving with companies investing in artificial intelligence (AI) and machine learning (ML) to bolster security capabilities. Hacking incidents are becoming the norm at the enterprise level, and companies are leveraging AI technology in response to these growing cyber threats. From Equifax, Hotel Marriot, NetEase, and Yahoo, these companies suffered major data breaches that exposed personal information and accounts. Social media companies have not been spared from hacking. Facebook, LinkedIn, and Chinese social platform Sina Weibo have come under attack from cyber criminals in the past.

Human intervention seems less effective today given the rise of malware attacks, phishing, and ransomware. The explosion of data and devices in the current digital environment has further escalated cybersecurity threats, giving cyber criminals more loopholes for attack. On average, a data breach costs a company close to $4M, considering that most enterprises detect threats in their systems after six months. These statistics paint a dim picture of the current cybersecurity landscape with most companies lacking a solid cybersecurity plan in 2021.

Figure 1

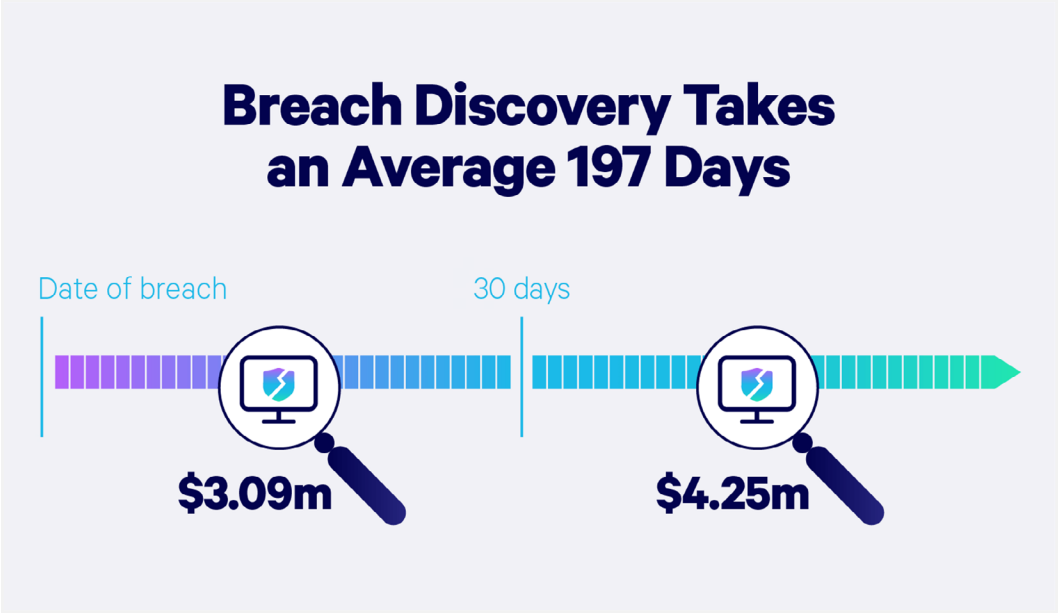


Image Source: https://www.embroker.com/blog/cyber-attack-statistics/

This is where artificial intelligence and machine learning can step in and take charge of enterprise security systems. Machine learning algorithms detect threats within a system, report them for prompt action, and reduce the time required to address cyber threats. A good example is a threat intelligence tool that utilizes ML capabilities for detecting cyber incidents. The precision of a system improves because of AI capabilities, which further improves threat detection.

Blockchain is coming of age in the current digital environment as enterprises leverage the technology, which prevents hackers from accessing sensitive information. The decentralized nature of blockchain makes it difficult for cyber criminals to attack the system because there is no single point of target. Because network traffic and storage of data are not in a single hub, security systems become stronger, giving hackers a hard time breaking through the database.

This article will explore application security predictions for artificial intelligence and blockchain, including threats, technologies, and practices.

Technologies and Practices

1. CRIME MANAGEMENT AND SECURITY

Public safety is critical to a functioning society, and while human intervention has not succeeded in reducing cybercrime, artificial intelligence is augmenting us by offering real-time information on system threats. The Avata Intelligence application in California is helping authorities predict and understand chances of terror attacks. By using game theory and artificial intelligence, the Avata Intelligence tool is bolstering the safety of California and enabling data-driven decision-making.

Smart drones are also trending with authorities using them for surveillance and mapping regions for crime evaluation. Drones cover large areas that the police cannot manage to oversee, and by relaying data in real time, cities around the world are using smart drones to safeguard against crime. The Armorway software used by the Coast Guard makes the surveillance of ports easier as the technology offers security status in Boston and New York. Port security is critical, and the Armorway software attests to the adoption of technology for improving security across different areas.

2. VULNERABILITY EVALUATION

Hacking incidents find companies least prepared — by the time intervention measures come up, the damage has already occurred. This gives hackers more power to keep exploiting company systems on servers and using loopholes to cause harm to data. Databases are using AI and machine learning to assess the vulnerability status and improve controls to prevent cyber-attacks. Database companies are using ML capabilities to secure client databases with advanced security measures.

A User and Event Behavior Analytics (UEBA) tool ensures that companies can review their system security across endpoints and servers. Anomalies within systems come in different forms; a UEBA tool — with the aid of artificial intelligence — ensures that enterprises detect these systemic issues early and offer intervention measures.

3. SECURITY OF DATA CENTERS

The data center infrastructure is critical for healthy functioning of enterprise security controls. AI technology is enabling optimization in areas like power consumption of data centers, cooling systems, rate of bandwidth, and backup systems. By offering insights into these areas of data center management, companies secure their systems by using analytics to understand the performance of data centers. Tech companies such as Google have achieved success by using AI for data center management, including the reduction of power and cooling expenditures.

4. INTERNET OF THINGS SECURITY

The Internet of Things (IoT) explosion has created new threats that organizations are using AI and ML to implement strengthened IoT security. AI predictive tools for companies such as AT&T are assisting in understanding customer sentiment and offering personalized services in real time. AT&T's use of predictive tools across their data centers has helped the company improve the customer experience and educate company leaders on their performance status across the board. The data processing at AT&T offers a glimpse of artificial intelligence and machine learning processes at work in enterprises committed to drive business growth by leveraging technology.

E-commerce companies are tapping into the power of AI to understand customer preferences and recommend products based on their previous shopping or search results.

Network analytics platforms are on the rise with a good example of IBM Watson for IoT, which offers real-time data on the condition of assets and provides solutions to improve the assets in bad condition. The demand for network analytics platforms will spike in the next decade as companies look to use AI for optimization of assets and cost management.

5. CONSUMER INFORMATION EVALUATION

The application of technology for consumer information management continues to evolve with companies using data to understand populations. The healthcare sector's use of data and personal information is rising, and this helps in understanding solutions for better health care. Despite concerns about invading personal privacy by using health records of patients, the healthcare industry represents another area ripe for technological innovation.

Thanks to the explosion of big data, it is becoming easier to understand populations by using social media platforms to gauge their health patterns and recommend data-based healthcare outcomes while respecting their privacy.

Threats

Artificial intelligence can improve security controls or necessitate vulnerabilities at the same time. This is a trend happening where hackers use AI technology in datasets to corrupt the data. Security systems come under attack in such an instance due to the manipulation of the learning system. Attacks also occur through transfer learning within ML systems because they are pre-trained. This gives hackers an opportunity to access systems because they understand the training models.

Training ML systems by use of manipulated inputs is growing as hackers develop their own training models. Hackers are going even further by training models again for system manipulation. Data privacy is at stake when hackers invade systems. This is a growing threat that could continue causing more harm if not addressed. Cyber criminals are developing effective mechanisms to infiltrate systems despite security control interventions.

Figure 2

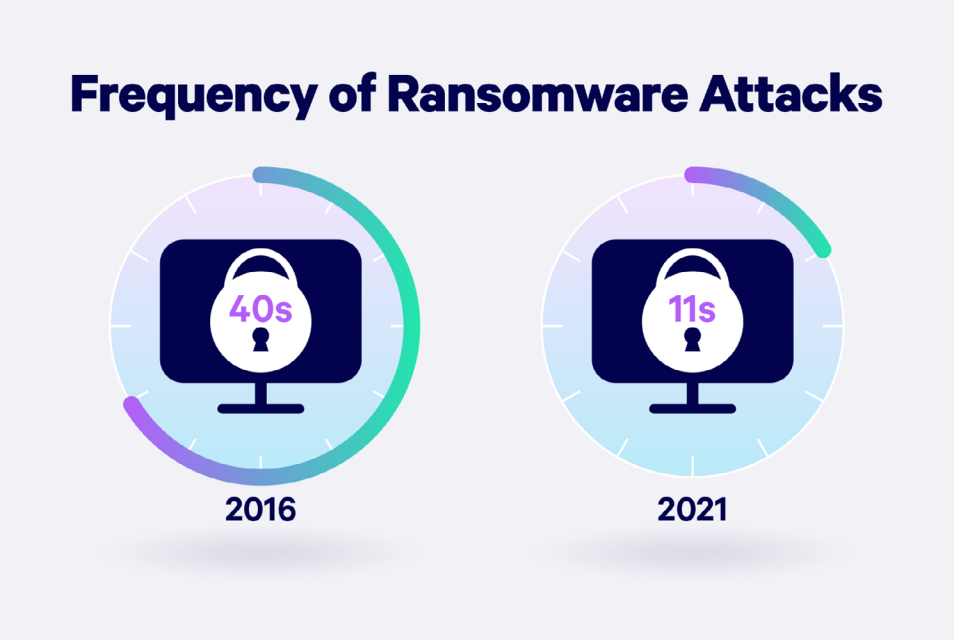


Image Source: https://www.embroker.com/blog/cyber-attack-statistics/

BLOCKCHAIN

Blockchain is proving an effective tool for cybersecurity because of the ledger technology that makes record-keeping public, thereby increasing security. Due to the decentralized nature of blockchain, detecting hacker activity is often quicker because the system training enables it to understand hacker attempts and threats, thereby reducing cybercrime.

Let us explore trends in application security predictions for blockchain in the next 12 months.

Data transmission creates attack windows for hackers, but progress is happening in blockchain technology with the development of encryption features. Encryption can prevent hackers from accessing data. In the next 12 months and beyond, companies will use blockchain and encryption to address the data transmission gap that often creates attack points for cybercriminals.

The Keyless Signature Infrastructure (KSI) offered through blockchain is another trend that addresses encryption problems by pinpointing weak points and safeguarding the system from external threats. KSI offers scalability for enterprises by building security infrastructures and improving solutions. The distributed nodes in blockchain make privacy easier — with the personal rights of users facing threats in the current situation, blockchain addresses this problem. The decentralization of data in blockchain allows personal information to be safeguarded, and this a win for individual privacy rights on data.

Contract management is another problem in the current security technologies environment where parties disagree over terms once a deal happens. Moreover, with cyber criminals who can manipulate contracts, the distributed ledger technology from blockchain will ensure that contracts uphold principles such as not reversible. Blockchain will solve this problem by fostering smart contracts, including both parties' tracking of contracts.

BLOCKCHAIN APPLICATION SECURITY USE CASES

MobileCoin is leveraging blockchain to enable companies to handle ledger information across networks in a simple way. Because most businesses have not transitioned to blockchain, MobileCoin is a company working to educate businesses on securing their ledgers. Coinbase implements blockchain through the storage of passwords, which requires users to provide security verification prior to accessing resources. This reduces fraud due to its strict encryption measures. Hashed Health is another organization adopting blockchain to combat security vulnerabilities, offering consulting services to enterprises on implementing blockchain technology with a focus on healthcare companies.

Lastly, the Australian government adopted blockchain technology by successfully securing public records through distributed ledger technology. Australia ranks among the top countries in the world to successfully adopt and implement blockchain technology for cybersecurity measures. More governments, including in Europe, are moving toward blockchain adoption.

Cybersecurity Systems with Artificial Intelligence and Blockchain

Data breaches in 2020 and in previous years remind us about the increasing need for security protocols to secure personal information and ensure businesses use data for the right use cases. With the privacy debate raging on every year, companies must pivot to AI and machine learning for system threat detection to consolidate network security.

Governments are also adopting security measures like Australia, using blockchain's distributed ledger technology. In the United States, the Transportation Security Administration (TSA) is using AI to bolster security systems across all airports by comparing data from passengers. Machine learning algorithms at the TSA are improving screening processes and reducing lines while facilitating crime detection and prevention. Additionally, the collaboration between individuals and machines in Europe through the Horizon Program enabled by artificial intelligence is a new era in security training.

Conclusion

Adopting a digital strategy starts with using AI and ML technologies for cybersecurity, real-time data processing, and ledger technology through blockchain. The traction toward blockchain as an alternative cybersecurity control of choice will spike in 2022 as companies become more data-driven. Businesses must create value within their operations — and securing vital company resources with state-of-the-art security practices and technologies is a good start.

David Yakobovitch, Host of HumAIn Podcast

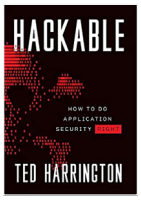
@dyakobovitch on DZone | @dyakobovitch on Twitter | www.humainpodcast.com

David Yakobovitch is the host of HumAIn Podcast — featuring Series A+ startups and Enterprise C-Suite on AI, data science, developer tools, and future of work. David is a Senior Manager at SingleStore, the database of choice for data-intensive applications. His role in Client Technical includes pre-sales, customer success, strategic operations, and education delivery. David is General Partner of DataFrame Ventures, a seed-stage syndicate that invests in data-powered startups. David previously scaled training programs on data science, data engineering, AI, and data analytics at Galvanize and General Assembly.

ADDITIONAL RESOURCES

Diving Deeper Into Application Security

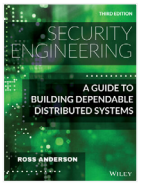
BOOKS



Hackable: How to Do Application Security Right

By Ted Harrington

No matter your role as a software professional, you are, in part, responsible for securing your application — for ensuring any security vulnerabilities are mitigated. Hear from a "leader of ethical hackers" in this book on AppSec, which covers how teams can better defend against attackers, from establishing a threat model to ingraining security throughout the SDLC.



Security Engineering: A Guide to Building Dependable Distributed Systems

By Ross Anderson

What does security mean today and beyond as our tech-connected world moves further toward cloud native? While many patterns of software misuse and exploitation remain largely the same, methods for AppSec continue to evolve. Learn about security psychology, attacker profiles and behavior, and practices for modern, sustainable security in the era of agile development, DevSecOps, cloud, and IoT.

TREND REPORTS

Containers

The mainstream shift toward cloud native introduces a new set of challenges that occur with a fundamentally altered software delivery pipeline, ranging from security to complexity and scaling. This Trend Report explores the common pain points of container adoption, principles for securing containerized workloads, and the broader impact of containers on overall application security.

DevSecOps

Security, like any other part of software development, is iterative — it takes rounds of testing and a keen focus to eliminate vulnerabilities. As more organizations realize the importance of security testing and integration, teams have started incorporating security into their DevOps pipelines. This report features insights from industry experts about the state of DevSecOps adoption and how to manage security throughout the SDLC.

REFCARDS

Getting Started With Static Code Analysis

Rather than execute program code to examine it, static analysis enables code review before testing begins, allowing teams to identify any defects, flaws, or vulnerabilities that may compromise the integrity or security of the application itself. In this Refcard, you'll explore the necessary tooling and steps for getting started with SAST, including practices for CI/CD integration.

OAuth Patterns and Anti-Patterns

Advancements of the OAuth spec along with its already comprehensive foundation can be challenging to understand, let alone put into practice. Securing access to APIs and other resources effectively under OAuth 2.0 requires first learning its key components and tools. This Refcard covers OAuth patterns, with a focus on security measures related to client credentials, OAuth flows, token validation, and more.

PODCASTS

Security Voices

Listen to the untold stories within the application security industry from insightful leaders who are making an impact while flying under the radar. Hosts Dave and Jack aim to "cut through the noise.... for 100% clear signal," showcasing these diverse voices from across the globe in 40+ episodes.

Application Security PodCast

Hosts Chris and Robert delve into AppSec topics ranging from security culture and DevSecOps to secure coding and OWASP with industry expert guests on each episode. Hear about the latest tools, practices, projects, and tips that will help you build a foundation for software security success!