OWASP Top 10 Risk for Citizen Development

Overview

Since Gartner introduced the term “Citizen Developer” in 2009, advances in platforms such as Microsoft Power Platform, SAP, ServiceNow, and Salesforce, along with AI assisted tools like Base44, Cursor, and Replit, have enabled people across all backgrounds to build sophisticated software at scale.

This movement has accelerated innovation but also raised new concerns around security and governance. The OWASP Citizen Development Top 10 Project addresses these challenges by identifying the most pressing risks that arise from citizen development of software built with Low Code No Code, AI assisted coding, and AI agent technologies.

Scope

Citizen developers can be business users improving efficiency in their daily work or hobbyists creating entirely new applications. By lowering barriers to entry, business innovation platforms and AI assisted technologies have redefined how software is created, removing long standing resource constraints from traditional development teams.

This expanded ability to create at scale introduces unique risks for organizations. Originally focused on Low Code No Code platforms, this OWASP project broadened its scope to reflect the wider landscape of citizen development and the technologies that enable it. The Top 10 risks therefore cover threats introduced through business innovation platforms, AI assisted coding, and AI agents.

Audience

The OWASP Top 10 Security Risks for Citizen Development is primarily aimed at security professionals who are tasked with enabling safe adoption of these technologies. They provide the critical guardrails that balance rapid innovation with secure practices.

The project also speaks directly to citizen developers, giving them an understanding of how their work may create risks and how to avoid common pitfalls. Governance bodies gain structured recommendations to align oversight with organizational goals and regulatory requirements, strengthening accountability across citizen development initiatives.

The key crossover focus for this document is to facilitate collaboration between the business and security and governance.

Mission

The mission of the OWASP Citizen Development Top 10 Project is to help organizations identify, prioritize, and mitigate risks tied to citizen development. The list explains how these risks manifest across various technologies and offers concrete mitigations to reduce them.

By providing accessible examples and guidance, the project equips both technical and non technical creators to make secure by default choices. Its goal is to establish the foundations that allow innovation to thrive while maintaining strong security practices.

Table of Contents

CD-SEC-01: Blind Trust

Description

Blind trust refers to the uncritical acceptance of outputs, recommendations, or artifacts produced by an automated system, platform, or model without independent verification of their accuracy, integrity, or security. In software development contexts—particularly AI-assisted and low-code/no-code environments—blind trust manifests when developers assume that system-generated code, templates, or components are inherently safe or correct, leading to overlooked vulnerabilities and systemic security risks.

Blind Trust in AI-assisted and low-code/no-code development should be treated as a fundamental operating assumption by security teams because it defines a critical risk profile. Citizen developers often lack security training and experience several cognitive biases during their development process. The first is automation bias where citizen developers treat platform generated code and pre-built components as inherently secure since they are readily available. Similarly, the second bias is the availability heuristic where easily accessible code is chosen over optimal code. Third, anchoring bias causes citizen developers to favor the template or design they are originally presented with for their solution, leading to the widespread adoption of insecure defaults and vulnerable marketplace templates. This unchecked development model, driven by psychological biases, leads to the widespread adoption of insecure defaults and vulnerable marketplace templates, which attackers can leverage to hide malicious code beneath the LCNC or AI-Assisted platform's abstraction. The lack of visibility and the ease of use create an environment where vulnerabilities accumulate undetected.

Example Risk Scenarios

Scenario #1

An attacker uploads a seemingly benign login template to a low-code/no-code platform's marketplace, disguised as a common and trusted solution. Unbeknownst to the platform or its users, this template contains a hidden, malicious script in a nested component or a custom code block. A citizen developer may trust the simplicity and appearance of the template and import the solution to build their own login page. When a user authenticates,

the hidden script will be triggered. The script will quietly capture the user's credentials and send them to the attacker's server without disrupting the normal login process. The citizen developer having blindly trusted the marketplace template and the platform's UI, would be completely unaware their solution is leaking credentials.

Scenario #2

A citizen developer is tasked with provisioning a new cloud storage resource, which requires defining a highly restricted Cloud IAM Policy adhering to the principle of least privilege. The developer asks the AI-assisted coding tool to generate the Infrastructure-as-Code (IaC) script (e.g., using YAML or HCL), explicitly requesting that the associated service account be granted read-only access to the resource.

The AI, attempting to apply complex conditional logic for security segmentation, hallucinates a syntax or module reference within the IaC script. The resulting configuration is syntactically correct and appears functional, but contains a logical security error: the generated Cloud IAM Policy defaults to an overly permissive permission set due to the failure of the hallucinated restrictive variables. The developer deploys the solution operating under blind trust, assuming the IaC script is secure simply because security controls were requested from the trusted AI source. An attacker can exploit this misconfiguration by compromising the service account, which, instead of being limited to read-only access, is granted full administrative control over the entire storage environment, leading to massive data exfiltration, modification, or deletion.

Scenario #3

A citizen developer, prioritizing speed, imports a seemingly professional solution from a company's marketplace in order to implement a login feature. Lacking the technical expertise to audit the code, the developer relies solely on the component's professional appearance and high user ratings . However, the solution may contain a hidden, malicious crypto-miner script nested deep within its logic. This script can secretly consume the company's backend server CPU/GPU cycles to generate cryptocurrency for the attacker. Because the login functionality works perfectly and is not visible through the low code / no code platforms user interface, it would go unnoticed for weeks.

Scenario #4

A citizen developer uses an AI assistant or low code / no code platform to build a solution that must interact with a third-party API. The generated code is syntactically secure, but lacks the necessary defensive programming to sanitize or validate incoming API responses and blindly trust the data received from the third-party. An attacker can exploit this flaw by injecting malicious data into the third party API response, compromising user data or system integrity.

Prevention – Blind Trust Secure Framework

OWASP's Top 10 Citizen Development project has created the "Blind Trust" Secure Framework, a strategic approach to the concept of blind trust structured around three core actions. Blind Trust is considered a unique and foundational risk category because it embodies and amplifies the consequences of all other security risks within the citizen development environment, such as insecure design, authentication failures, and logging issues. Due to this comprehensive nature, it has been given a designated risk framework that the other listed risks do not have. Security teams can incorporate this specialized framework into their risk assessment to effectively address developer bias and turn it into a security asset.

Enforce Secure Defaults

This prevention strategy addresses the fundamental behavioral biases of Citizen Developers towards simplicity, speed, and immediate access. This aims to recognize that developers often choose the path of least resistance, and design workflows that make the secure option the easiest and most convenient choice.

- Design platforms with secure-by-default configurations, ensuring that default settings are inherently safe and difficult to disable.
- Establish a standardized and vetted component library so developers can safely reuse trusted code and templates.

Extend the Scope of Governance

This strategy addresses the disconnect between our classifications of citizen developed apps. We extend the scope of security by treating citizen development apps as governed extensions of our official environment, moving beyond the core platform's API to ensure security controls and continuous, real-time oversight extend into the configuration and data access of every single application.

- Integrate these controls into a governance framework that enforces compliance and ensures consistent application of security policies across all AI-assisted and low-code/no-code projects.

Secure Innovation Through Trust

We secure the innovation through trust at every stage by shifting security into an integrated component of application quality throughout the entire development lifecycle.

- Provide just-in-time security guidance embedded directly within the development environment to reinforce secure decision-making during coding and configuration.
- Offer structured training resources and reference guides to improve citizen developers' understanding of common vulnerabilities and secure development practices.
- Implement a pre-deployment security auditing and validation system to automatically review all generated or modified code before release.

References

- Halley, M. (2023, November 28). The privacy and security risks of "citizen development". Technology First. https://www.technologyfirst.org/Tech-News/13284443#:~:text=Ensure%20that%20citizen%20developed%20applications,to%20design%20security%20into%20applications.
- Rowntree, S. (2024, July 3). The perils of blind trust in AI – why human accountability must not be automated away. Oliver Wight EAME. https://oliverwight-eame.com/news/the-perils-of-blind-trust-in-ai-why-human-accountability-must-not-be-automated-away

- Savunen, T., Kekolahti, P., & Mähönen, P. (2023, May 17). Blind trust in enhancement technologies encourages risk-taking even if the tech is a sham. Aalto University. https://www.aalto.fi/en/news/blind-trust-in-enhancement-technologies-encourages-risk-taking-even-if-the-tech-is-a-sham
- Rowntree, S. (2025, August 18). Training alone can't fix the citizen developer security gaps. TechNewsWorld. https://www.technewsworld.com/story/why-training-wont-solve-the-citizen-developer-security-problem-179877.html

CD-SEC-02: Account Impersonation

Description

Citizen developed applications often blur the line between an application's own identity and identities of the people or services it connects with. Instead of using unique, well-governed application accounts, creators may embed personal credentials, shared service accounts, or database connections to enable quick access to data and services. These embedded identities might belong to the developer, a team, or an external system and are then implicitly used by anyone interacting with the application.

Because the application operates under these embedded identities, security and monitoring systems typically see all activity originate from the same user, making every action to be from the creator or shared account. This allows any user of the application to impersonate its creator, concealing the true actor and creating a direct path for privilege escalation. This risk is amplified when a single application spans across multiple environments or platforms.

Code agents can produce authentication or session logic that appears functional but lacks secure mechanisms such as unencrypted cookies or weak credential storage. They may even create or reuse service accounts without proper governance leading to inconsistent role enforcement unmonitored credentials. In such environments, organizations face hidden actions, broken audit trails, and increased liability as malicious actors or even unknowing users can impersonate others, bypass security controls, escalate privileges undetected.

Example Risk Scenarios

Scenario #1

A citizen developer creates a simple application to view records from a database to accomplish a business task. They use their identity to log into the database, creating a connection embedded within the application. Every action that any user performs in this application ends up querying the database using the developer's identity. A malicious user

takes advantage of this and uses the application to view, modify or delete records they should not have access to. Database logs indicate that all queries were made by a single user, the trusted developer.

Scenario #2

A developer creates a business application that allows employees to fill out forms with their personal information. To store form responses, the developer uses their personal email account. Users have no way of knowing that the app is storing their data on the developer's personal account.

Scenario #3

A developer creates a business application and shares it with an administrator. The developer configures the app to use its user's identity. Aside from its stated purpose, the app also uses its user's identity to elevate the privileges of the developer. Once the admin uses the app, they inadvertently elevate the developer's privileges.

Prevention

- Adhere to the principle of least privilege when provisioning connections to databases/services/SaaS
- Leverage OAuth for access control with user's consent to the resource server
- Restrict the use of "creators identity" when an application is designed to be shared. Use a dedicated service account instead and apply dedicated behavior monitoring to those service accounts.
- Monitor for apps that request highly privileged OAuth scopes unrelated to the apps purpose
- Ensure a proper audit trail is maintained to identify the actor behind actions performed through the shared connection, whether those connections are shared by virtue of users using the application or by granting users access to that connection directly

References

- Bargury, M. (2022, July 18). Watch out for user impersonation in low-code/no-code apps. Dark Reading. https://www.darkreading.com/cyber-risk/watch-out-for-user-impersonation-in-low-code-no-code-apps
- Bargury, M. (2022, June 20). Credential sharing as a service: The hidden risk of low-code/no-code. Dark Reading. https://www.darkreading.com/cyber-risk/credential-sharing-as-a-service-hidden-risk-of-low-code-no-code
- Rubinstein, D. (2022, July 26). Do low-code/no-code platforms pose a security risk? SD Times. https://sdtimes.com/lowcode/do-low-code-no-code-platforms-pose-a-security-risk/
- Zenity. (2022, July 18). Watch out for user impersonation in low-code/no-code apps [Video]. YouTube. https://www.youtube.com/watch?v=D3A62Rzozq

CD-SEC-03: Authorization Misuse

Description

Applications rarely operate in isolation, and this remains true for those built by citizen development platforms. It is standard for these applications to integrate across an organization's technology stack, connecting to SaaS, PaaS, cloud, or on-premise systems. To enable this ability, these technologies typically rely on pre-built connectors, API wrappers, or embedded authentication methods to allow quick setup and reuse of credentials.

The risk arises when authentication and authorization are handled in an overly permissive, opaque or overly complicated way. Due to these perceived blockers, citizen developers may unknowingly or carelessly embed personal credentials or use access tokens to speed up integration. OAuth flows, refresh tokens, and API keys are often stored or shared across applications and persist well beyond their intended lifecycle. A quickly established credential may remain active indefinitely, accessible to other users or applications, and used for purposes outside of the original intent.

Additionally, authorization scopes are frequently defined too broadly. Instead of restricting access to only the minimum required resources, applications are granted expansive permissions for the sake of flexibility or convenience, meanwhile exposing sensitive assets to unnecessary risk. Gradually, this leads to an ecosystem of abandoned, or "zombie" connections. These connections are long-lived, broadly scoped, and challenging to monitor or revoke, making them primed for malicious activity or unintentional data exposures.

In summary, authorization misuse in citizen-developed technologies stems from a combination of ease-of-use features, poor credential hygiene, and a preference for broad persistent access. This risk applies equally to low code/no-code builders, AI-driven application generation, and other rapid development approaches where speed takes precedence over security.

Example Risk Scenarios

Scenario #1

A citizen developer uses a low-code platform to quickly build an app that pulls sales data from their customer management system and pushes updates into a spread sheet shared across their team. To simplify setup, the analyst grants the app “read and write all data” access in the customer management system rather than limiting it to just sales reports. The platform stores their OAuth refresh token indefinitely, and the connection is reused when another team member clones the app for a different purpose. Months later, when the analyst changes roles, their Salesforce account is still the one powering several workflows, meaning others can still use their credentials. An attacker finds this exploit and uses the broad authorization to modify or exfiltrate sensitive Salesforce records, well beyond what the analyst ever intended.

Scenario #2

A citizen developer asks an AI coding assistant to generate a script that automates uploading customer leads from emails into the company’s CRM. The developer provides their personal API key to the AI assistant, which hardcodes it directly into the generated script. The script requests “admin” access to the CRM API because it was the quickest way to avoid permission errors. The script is then shared with the broader team. The hardcoded admin key is copied, stored, and reused by multiple employees. A rogue employee eventually finds this key and uses it to gain full administrative access to the CRM.

Scenario #3

Admin connects an application to their source code management system (e.g., Bitbucket) using a service or application account. The provisioned service or application account has unrestricted access to all repositories to enable seamless integration. Any internal user can abuse this connection to access restricted repositories they usually don’t have access to.

Prevention

- Disable or monitor the use of implicitly shared connections
- Adhere to the principle of least privilege when providing access to environments that can contain shared connections
- Monitor citizen development platforms for over-shared connections
- Educate business users on the risks of connection sharing and its relation to credential sharing
- Explicitly refresh OAuth tokens on a regular basis by re-authenticating connections.
- Carefully review the scope an application requires and adhere to the principle of least privilege

References

- Bargury, M. (2022, June 20). Credential sharing as a service: The hidden risk of low-code/no-code. Dark Reading. https://www.darkreading.com/cyber-risk/credential-sharing-as-a-service-hidden-risk-of-low-code-no-code
- Bargury, M. (2022, May 12). Why are low-code platforms becoming the new holy grail of cyberattackers? Zenity. https://www.zenity.io/blog/why-are-low-code-platforms-becoming-the-new-holy-grail-of-cyberattackers/

CD-SEC-04: Sensitive Data Leakage and Handling Failures

Description

Citizen development platforms are powerful tools for connecting systems and automating processes, but these conveniences often come at the cost of security. Both platforms and their users often lack the necessary context and controls for safe data handling, resulting in two critical risks: uncontrolled data leakage and cascading operational failures.

Data leakage occurs when sensitive information is transmitted beyond its intended storage location or outside the organization's control. Most citizen development tools cannot automatically classify, secure, or track sensitive data throughout an application's lifecycle. Designed for flexibility rather than security, these platforms lack understanding of the context or purpose of the data they process and therefore cannot identify or tag sensitive fields. As a result, citizen developers may unintentionally expose data through misconfigured permissions, unprotected connections, public storage, unsecured APIs, or AI-generated logic that transmits or logs information without encryption. The absence of proper data governance on both the platform and user sides allows sensitive information to be collected, stored, and shared without visibility, tracking, or protection.

Operational risk also increases when non-technical users connect disparate systems through citizen development workflows. These connections often create undocumented dependencies, forming a fragile web where small changes can trigger cascading effects across multiple processes. Because these operations are rarely mapped or monitored, they can cause disruptions, outages, and data integrity issues. Ultimately, these risks highlight how speed and convenience in citizen-developed applications often come at the expense of security and governance.

Example Risk Scenarios

Scenario #1

A sales operations analyst uses a low code citizen development platform to automatically sync customer data from the CRM into the marketing database for campaign tracking. The sync workflow includes sensitive customer PII not required for marketing use. The marketing database is hosted in a SaaS platform with weaker controls. A change in the CRM schema can trigger multiple downstream apps to mis-map fields and cause corrupted campaign data. Sensitive PII will be exposed inappropriately and data quality errors can be spread across customer-facing campaigns.

Scenario #2

A product manager uses an AI assistant to generate a script that analyzes support tickets and sends summaries to the internal messaging platform for visibility. The manager pastes raw support logs which contain customer account IDs and occasional confidential error messages into the AI assistant prompt. The assistant may generate code that sends detailed summaries to a public Slack channel with 200+ employees. Confidential data can leave its secure support environment and be broadcast to employees who do not possess privileged access to this information, creating data leakage and compliance risks.

Scenario #3

A citizen developer uses a low code platform to create an internal application that retrieves data from a private cloud storage bucket. To make the data available for internal dashboards, the developer creates a public facing API endpoint in the application that serves the data. The developer accidentally misconfigures the APi endpoint to return the data in a raw, un-sanitized format, which includes the access credentials for the cloud storage bucket in the response body. An attacker can discover this public API endpoint and send a curl request to retrieve the hard-coded storage creds in the response body, compromising the private cloud storage bucket.

Scenario #4

An application is created by a citizen developer in a low code/no code platform and connects to an internal database. For simplicity, the database connection string, which includes sensitive credentials, is stored in a variable within the app's environment. The platform encrypts this value and in its live runtime, giving a false sense of security to the developer. When a developer exports this component to share, the platform fails to scrub this data from the export. The connection string, containing the plaintext password, is included in the exported file. A bad actor may find this password in a shared drive or public marketplace and use the found credentials to connect to the internal database directly, bypassing the platform's security and gaining full access to the company's data.

Scenario #5

A citizen acting as an application developer uses an AI assistant to create a webhook listener. The AI, acting as a functional but unthinking tool, provides code that logs all incoming data without sanitizing it. Unaware of the security risk, the developer deploys this code. An attacker who has gained basic access to the logging functionality can discover a sensitive client secret from a partner's webhook payload and use this information to impersonate the partner's service and manipulate the application.

Prevention

- Limit connectors to an approved services list
- Limit creation of custom connectors to dedicated personnel
- Monitor platforms for data flow outside the organizational boundary, including multi-hop paths
- Work with the business to understand the citizen development needs and provide simple alternatives that meet security requirements
- Educate business users on the compliance, privacy, and security risks related to data storage
- Monitor managed databases, environment variables, and configuration provided by citizen development vendors for sensitive data

- Ensure security teams are involved in security reviews with applications having access to sensitive data
- Enforce secure standardized data transfer between combined citizen development workflows

References

- Bargury, M. (2022, August 29). 3 ways no-code developers can shoot themselves in the foot. Dark Reading. https://www.darkreading.com/dr-tech/3-ways-no-code-developers-can-shoot-themselves-in-the-foot
- Bargury, M. (2021, October 12). Hackers abuse low-code platforms and turn them against their owners. Zenity. https://www.zenity.io/blog/hackers-abuse-low-code-platforms-and-turn-them-against-their-owners/
- Bargury, M. (2022, January 16). Low-code security and business email compromise via email auto-forwarding. Zenity. https://www.zenity.io/blog/low-code-security-and-business-email-compromise-via-email-auto-forwarding/
- Microsoft Detection and Response Team. (2020, April 2). Full operational shutdown—another cybercrime case from the Microsoft Detection and Response Team. Microsoft Security Blog. https://www.microsoft.com/en-us/security/blog/2020/04/02/full-operational-shutdown-another-cybercrime-case-microsoft-detection-and-response-team/
- Microsoft. (2025, September 3). Implement a data policy strategy. Microsoft Learn. https://learn.microsoft.com/en-us/power-platform/guidance/adoption/dlp-strategy

CD-SEC-05: Authentication and Secure Communication Failures

Description

Citizen-developed technologies frequently handle connections to sensitive systems and data. These connections rely on authentication methods and secure communication protocols to protect data in transit and ensure only authorized users or applications gain access.

However, because citizen developers are often more focused on functionality than security, authentication, and communication are commonly misconfigured. Common security flaws include the use of weak or personal credentials, storing secrets or tokens in plaintext, disabling TLS/SSL certificate validation, and selecting insecure communication protocols (e.g., HTTP instead of HTTPS). In some cases, authentication flows are only partially implemented or improperly applied. A few examples of this include omitting multi-factor authentication (MFA), embedding long-lived bearer tokens directly in source code, or reusing static credentials across multiple applications or environments.

Misconfigured or insecure integrations can enable unauthorized access, facilitate credential compromise, and allow interception or manipulation of data in transit. Over time, this leads to an accumulation of ungoverned, weakly secured connections that violate corporate security policies and remain difficult to detect or remediate. Ultimately, authentication and secure communication failures occur when citizen-developed technologies fail to generate or enforce secure-by-default configurations, allowing insecure connection practices to persist.

Example Risk Scenarios

Scenario #1

A customer support manager builds an LCNC app to pull ticket data from a customer support service and push summaries into a dashboard. The connection is set up using the manager's personal username/password instead of an API key with scoped permissions. TLS validation is disabled to avoid repeated connection errors. The same connection is reused by multiple apps across the team. If the manager's password is compromised, attackers gain broad access to Zendesk. Additionally, disabling TLS exposes data in transit to interception.

Scenario #2

A citizen developer asks an AI coding assistant to generate a script that connects a financial system API to an internal reporting tool. The assistant generates code that hardcodes API keys directly into the script without encryption or secret management. It defaults to HTTP instead of HTTPS for the API endpoint. The developer copies the script into a shared GitHub repo visible to contractors. The exposed API keys and insecure transport channel create both authentication and communication risks, allowing unauthorized third parties to access sensitive financial data.

Scenario #3

A developer creates an application that uses an FTP connection but does not check the box that turns on encryption. Users of that app have no way to know that their data is being transferred unencrypted since the communication between the app and its users is encrypted.

Scenario #4

A developer uses administrator credentials to create a database connection. They build an application that uses that connection to show data to its users. Even though they intended to allow read-only operations through the app, users can use the over-privileged connection to write or delete records from the database.

Prevention

- In production environments, limit the creation of any new connections to dedicated and authorized personnel or teams.
- Monitor platforms for connections that do not comply with best practices
- Create secure authentication framework code and configuration settings or solution component templates for easy implementation of secure authentication, authorization, and communication
- Provide training for citizen developers on the risk of insecure communication

References

- O'Connor, C. (2020, February 19). CISO View Insights: Securely scaling RPA initiatives. CyberArk. https://www.cyberark.com/resources/blog/ciso-view-insights-securely-scaling-rpa-initiatives

CD-SEC-06: Vulnerable and Untrusted Components

Description

When dealing with complex workflows in citizen development solutions, there's a significant risk that nested logic will obscure security vulnerabilities. For example, a main workflow may rely on sub-workflows or pre-built components to perform specific actions. These sub-processes may have different or entirely missing security controls compared to the main workflow, and the citizen development platform often lacks the tools to provide a clear, unified view into these security gaps, inherently trusting that the underlying functionality is safe.

This risk is compounded when developers use a marketplace to share and reuse these components. An insecure sub-workflow or component, once created and uploaded to the marketplace, can be easily downloaded and integrated by other developers, causing a single security flaw to be replicated and propagated throughout the entire organization's applications. This creates a supply chain vulnerability, where the platform solutions are subject to the trust of unvetted components from the developer community or organization.

AI coding assistants, in an effort to provide consistency, may suggest or reuse a previously generated code block that contains a security flaw. If a developer accepts and deploys this flawed code in multiple parts of an application, the single vulnerability becomes a widespread issue throughout the codebase. This practice, often a result of over-reliance on AI, can introduce a consistent and pervasive weakness that is difficult to remediate, as every instance of the reused code must be identified and patched.

Crucially, AI assistants may also "hallucinate"packages, libraries, or APIs that do not actually exist or that reference outdated/insecure versions. If a citizen developer includes one of these non-existent or flawed dependencies in their application based on the AI's suggestion, it can lead to build failures, runtime errors, or the introduction of a critical, unverified security weakness that is extremely difficult to trace back to a legitimate source. This inherent risk of AI-generated misinformation adds a new, complex layer to the existing supply chain security problem.

Example Risk Scenarios

Scenario #1

A citizen developer creates a simple subflow to publish a status update to a webhook for Business Intelligence ingestion. One solution publishes business relevant status updates, while another one unknowingly includes user PII which is propagated outside of the system.

Scenario #2

A citizen developer integrates a highly rated component from a low code / no code platform marketplace into a critical, customer facing application. This widely used component contains a hidden server side request forgery vulnerability within its data filtering logic. An attacker can recognize this component's ubiquity and exploit this flaw across multiple applications. The popularity of a flawed component utilized for convenience would become a widespread security crisis.

Scenario #3

A citizen developer using an AI Assistant to code to build an application, receives a hallucinated reference to a non-existent package. Due to the blind trust in the AI's output, the developer searches for the suggested component and finds an unknown package with the exact name. The developer installs the phantom package which contains a preinstalled malicious script. The attacker who owns this package may now exfiltrate sensitive information from the developers application.

Scenario #4

A citizen developer asked their AI code assistant to generate the repetitive boilerplate for form input validation across an application. The AI, however, created a code block containing an exploitable Cross-Site Scripting vulnerability. Driven by a desire for consistency, the developer reused this exact, insecure code for every form in the application. This action scaled a single oversight into a systemic flaw, offering an attacker numerous XSS attack vectors and effectively turning the application's uniform architecture into a flawed blueprint for widespread compromise.

Prevention

- Ensure each component adheres to the dedicated security controls utilized by the core application
- Clearly document what the sub workflow expects as input and output, and what data should be validated beforehand.
- Create standardized and secure subcomponents for common/redundant functionality to facilitate easy access and usage

References

- Mendix. (2023, October 5). Incident 8j5043my610c. https://status.mendix.com/incidents/8j5043my610c
- OWASP Foundation. (2021, September 15). A06:2021 – Vulnerable and Outdated Components. https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/
- Zenity. (2022, May 12). Why are low-code platforms becoming the new holy grail of cyberattackers? https://www.zenity.io/blog/why-are-low-code-platforms-becoming-the-new-holy-grail-of-cyberattackers/

CD-SEC-07: Security Misconfiguration

Description

Low-code/no-code development platforms provide a wide range of features, some of which control the balance between security and support of specific use cases. However, AI-assisted coding, in its pursuit of functional and complete code, can disrupt this balance by generating or suggesting code that lacks critical security features.

A citizen developer who relies on this AI-generated code without sufficient security expertise may unwittingly introduce security misconfigurations. This can result in anonymous user access to sensitive data, unprotected public endpoints, exposed secrets, and oversharing. This risk is compounded by the fact that many configurations are set at the application level rather than the tenant level, giving business users themselves, rather than administrators, the authority to implement these insecure defaults.

AI-generated code may lack critical security features because of a fundamental conflict between the AI's design and the citizen developer's needs. An AI coding assistant's primary goal is to provide a functional and direct response to a prompt. It excels at literal interpretation and aiming for exactness. If a citizen developer asks for a function like "create a login form that checks a password," the AI will provide a solution that does exactly that. It won't, however, automatically include security best practices that weren't explicitly requested.

The risk is that the AI acts as a skilled but unthinking tool. It prioritizes delivering a "working" solution over a "secure" one, and it lacks the security context to know that its literal output could introduce a vulnerability. This is especially dangerous when a developer assumes the AI's code is safe by default, leading to the deployment of insecure code that is functionally correct but easily exploited.

Example Risk Scenarios

Scenario #1

A citizen developer uses a low code / no code platform to build a public facing web form. The platform automatically generates a REST API endpoint to process the form data. For developer convenience, the platform's default configuration for this API is "publicly accessible without authentication". An attacker can discover the API endpoint of this application through a network scan. Given the endpoint's default configuration, the attacker can send malicious requests to the API and cause the server to expose sensitive customer data all without authentication.

Scenario #2

A citizen developer uses an AI assistant to generate code for a logging function. The AI provides code that is correct for storing logs but includes a default setting that makes the storage location publicly accessible on the internet. The citizen developer, not being a security expert, deploys this code without recognizing the insecure default. An attacker can discover these public logs and steal sensitive information, including API keys or user tokens. This can occur not because the code was broken, but because the AI output was insecure by default.

Scenario #3

A citizen developer builds a new application using a low code/no code platform that requires a password to be set by a user. The "rules" configured for this new application are less secure than the company's password policy. The less secure password policy can make it easier for an attacker to guess the user's password and access the system.

Scenario #4

An application is created by a citizen developer that allows users to upload documents to a server. The upload process has not been configured to scan the document upon upload for security risks. A malicious user can use this process to upload a document containing malware and lock all computers in the company unless payment is made resulting in a ransomware attack.

Prevention

- Build tools for checking generated code for hard coded and insecure configurations
- Create securely configured templates to encourage best practices while adhering to customization
- Build validation functionality for solutions to detect issues before deployment
- Monitor configuration for drifts from secure patterns
- Read citizen development platform vendor documentation and follow recommendations
- Ensure configurations align with industry best practices
- Implement a change management system for tenant-level configuration

References

- UpGuard. (2021, August 23). By design: How default permissions on Microsoft Power Apps exposed millions. https://www.upguard.com/breaches/power-apps
- Zenity. (2022, May 12). Why are low-code platforms becoming the new holy grail of cyberattackers? https://www.zenity.io/blog/why-are-low-code-platforms-becoming-the-new-holy-grail-of-cyberattackers/
- Zenity. (2022, May 12). Why are low-code platforms becoming the new holy grail of cyberattackers? [Video]. YouTube. https://www.youtube.com/watch?v=e8PEIOa6W9M
- Microsoft. (2025, February 18). Reference architecture and landing zones for Power Platform. https://github.com/microsoft/industry/blob/main/foundations/powerPlatform/referenceImplementation/readme.md#power-platform-landing-zones-reference-implementation

CD-SEC-08: Injection Handling Failures

Description

The risk of injection handling failures is a critical security concern in low-code/no-code and AI-assisted solutions. This risk arises when an application ingests user-provided data from various sources, but fails to properly sanitize or validate it before using it in a command or query. This is a significant problem because many citizen development applications are designed to dynamically query data based on user input. As a result, they are highly exposed to injection attacks. The risk is compounded by a lack of security context: neither the AI nor the average citizen developer understands what makes data sensitive or dangerous. This can lead to code that is functionally correct but easily exploited.

The AI, in its pursuit of a "working" solution, may not always parameterize queries, which allows attackers to inject malicious code. Many citizen development platforms also lack the ability to vet input schemas for possible injection, making them vulnerable by default. Ultimately, this can lead to the deployment of insecure solutions that are functionally correct but easily exploited. Low-code/no-code platforms often use a specific, non-standard syntax to reference variables and platform data within a solution. This creates a unique form of injection vulnerability. The risk arises when an LCNC application ingests user-provided data but fails to properly sanitize it for this special syntax. An attacker can then inject a string that is interpreted as a command to retrieve sensitive, internal platform data, such as a company's organization ID, even if they are not authenticated. The attack succeeds not because of a flaw in the developer's code, but because of a vulnerability in the platform's own core engine. The platform's lack of input sanitization for its custom syntax creates an injection vulnerability that bypasses all of the developer's security controls.

Example Risk Scenarios

Scenario #1

A citizen developer uses a low code / no code platform to create a simple online form. The form is designed to take a user’s input and simply show it back to them on the screen. The

platform contains its own special internal development syntax for easy data reference within the application and is used to easily reference the form's data and output it. Because this syntax is specific to the platform, it is not accounted for in common security input sanitization mechanisms. An attacker can discover this form and enter a command using the platform's internal development syntax to reveal company information stored within the application and other sensitive data.

Scenario #2

A citizen developer sets up an automation script using an AI assistant that triggers every time a new publication is made to an RSS feed and stores it into a SQL database. An attacker controlling the feed can use this flow to inject commands into the database and delete important records.

Scenario #3

A citizen developer creates an application that allows users to fill out forms. The app encodes form data as CSV files and stores them on a shared drive. Even though the platform sanitizes form inputs for SQL injection attacks, it does not sanitize for Office macro attacks. An attacker can take advantage of this oversight and input a macro that gets written into the CSV file and executes when a user opens the file.

Prevention

- Sanitize user input, taking into account the operations that will be performed on that input by the application
- For database interaction via SQL, also use query parameterization, stored procedures, or escaping
- Educate business users on the risk of unsanitized user input. Platforms cannot make this problem go away on their own
- Ensure platform syntax is accounted for in sanitization efforts on a platform level
- Sanitize platform specific syntax to avoid targeted injections
- Use rules and system prompts to instruct code agents to generate input validation logics for form and user input processing, and manually validate generated code.

Reference

- Aabashkin, A. (2022, May 12). Security vulnerabilities in robotic process automation (RPA) / low code technology: SQL injection & the return of Bobby Tables. Retrieved from https://aabashkin.github.io/posts/rf_sqli#a-modern-example-with-a-low-code--rpa-platform
- Open Web Application Security Project (OWASP). (2021). A03:2021 – Injection. Retrieved from https://owasp.org/Top10/A03_2021-Injection/
- Salesforce. (n.d.). SOQL injection security tips. Retrieved from https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/pages_security_tips_soql_injection.htm

CD-SEC-09: Asset Management Failures

Description

The core principle of citizen development platforms is to enable rapid application with seemingly low maintenance overhead. This ease of development combined with the volume and custom configuration of citizen developed solutions, leads to ungoverned, unpatched, and ultimately orphaned solutions. Because citizen developed applications are easy to create and often rely on SaaS services for management, organizations quickly see the number of active solutions grow. This agility contributes to the critical risk of asset management failures. Citizen developed applications are prone to being forgotten or abandoned while remaining active and functional. Furthermore, useful internal applications become widely shared within the organization, scaling rapidly in usage and dependency without the necessary business continuity measure expected of more standardized and essential systems such as having designated owners, IT monitoring, or a Service Level Agreement (SLA). The challenge of maintaining a clear inventory becomes nearly impossible due to the sheer volume.

The uniqueness and variety of these citizen developed solutions and workflows create a significant problem for version control and security patching. Since citizen development solutions are often widely shared or custom configured, they are not patched easily when a flaw is discovered in an underlying component. This leaves the process of securing existing, at-risk applications highly manual and unreliable. Moreover, the lack of central tracking or version control for these applications means security flaws are often undetectable. Without proper channels for mitigation, an organization has decreased visibility into its solutions, increasing its attack surface and maintenance debt.

Example Risk Scenarios

Scenario #1

A highly used low code/no code application becomes orphaned. This app is outside of IT's monitoring and patching schedule. An attacker can easily exploit a known vulnerability in this application since it is unpatched. This foothold can be used to access and steal sensitive data from the connected services, leveraging the application's established but unreviewed permissions. Since the application is unmonitored, this data leakage could go undetected for an extended period of time.

Scenario #2

A citizen developer uses AI-assisted coding to build a critical business workflow deploying it as shadow IT without a security review. The developer inadvertently incorporates a malicious dependency from a compromised public code source suggested by the AI. This malicious code is deployed undetected, bypassing all governance. When activated, it exploits the workflow's permissions and its implicit coupling to inject data-stealing routines into other connected, popular applications and successfully can exfiltrate information because the solution lacks proper version control and central oversight.

Scenario #3

A citizen developer browser through the platform marketplace for a low code / no code solution and explores application templates. Each click creates an app with an external-facing URL. The developer forgets about these apps, even though they might expose business data. This scenario multiplies by the number of developers, resulting in an ever-growing number of stale apps.

Prevention

- Establish a centralized governance center
- Enforce application ownership and lifecycle
- Implement tiered risk classification
- Mandate version control and secure deployment
- Maintain a comprehensive inventory that lists applications, components, and users

- Remove or disable unused dependencies, unnecessary features, components, files, and documentation

References

- Bargury, M. (2022, August 29). 3 ways no-code developers can shoot themselves in the foot. Dark Reading. https://www.darkreading.com/dr-tech/3-ways-no-code-developers-can-shoot-themselves-in-the-foot
- Bargury, M. (2022, May 16). You can't opt out of citizen development. Dark Reading. https://www.darkreading.com/cyber-risk/you-can-t-opt-out-of-citizen-development
- Bargury, M. (2022, March 1). Why so many security experts are concerned about low-code/no-code apps. Dark Reading. https://www.darkreading.com/dr-tech/why-so-many-security-experts-are-concerned-about-low-code-no-code-apps

CD-SEC-10: Security Logging and Monitoring Failures

Description

Citizen developed applications often fail to establish secure and compliant logging practices and fall into two polarized modes. On one end, there will be an absence of rigorous logging and the platform will lack audit capabilities found in traditional software development. The underlogging gap creates significant challenges for security and compliance. When critical actions are not captured, organizations lose the ability to reconstruct events during an incident or breach accurately. Insufficient logging leaves investigators blind to the root cause of a compromise, while excessively or poorly secured logs can expose confidential information and violate privacy regulations.Instead of implementing a clear, end to end audit trail, these applications rely on platform defaults or ad hoc logging implemented by citizen developers. This means repudiation for these applications is left incomplete, inconsistent or altogether absent.

On the opposite end, over-logging for troubleshooting as a common feature of the development process. Overlogging is symptomatic of the focus on rapid iteration and debugging and occurs when platforms provide comprehensive logging settings to gain deep visibility into application behaviors. Citizen developers may enable logging settings meant for debugging but, in the process, capture sensitive data meant for runtime activities only. While this extensive logging aids in development and initial testing, when left active in production or poorly secured, it can expose confidential information or violate privacy regulations.

The problem is exacerbated by the diverse and often experimental nature of citizen development. Applications may integrate multiple services, switch environments, or be generated by AI with minimal human review, further fragmenting and obscuring the audit trail. Without a reliable record of who made changes, what actions were performed, and when they occurred, organizations face heightened risk of undetected malicious activity, failed compliance audits, and prolonged recovery times after an incident.

Example Risk Scenarios

Scenario #1

Application logs are turned off. When a breach attempt occurs, security teams are unable to determine who accessed the app and what they tried to do.

Scenario #2

A business-critical application stops functioning following a change. Since multiple changes have occurred, each resulting in an application update, it is challenging to find which developer introduced the particular change that caused the issue. Developers have to review each application version manually to locate the problematic version. Since each application "save" translates to an update, the number of updates would make a manual process prohibitively expensive and time-consuming. On some platforms, developers can only review the application's current version, so they won't be able to find or revert to a stable version.

Prevention

- Leverage platform built-in capabilities to collect user access and platform audit logs
- Where applicable, instrument applications with logging mechanisms to provide extra visibility
- Ensure logs are not contaminated with sensitive data by configuring the platform to avoid logging raw application data

References

- Microsoft Detection and Response Team. (2020, April 2). Full operational shutdown—another cybercrime case from the Microsoft Detection and Response Team. Microsoft Security Blog. https://www.microsoft.com/en-us/security/blog/2020/04/02/full-operational-shutdown-another-cybercrime-case-microsoft-detection-and-response-team/

- OWASP Foundation. (2021). A08:2021 – Software and data integrity failures. Open Web Application Security Project. https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/
- Zenity. (2022, May 12). Why are low-code platforms becoming the new holy grail of cyberattackers? Zenity. https://www.zenity.io/blog/why-are-low-code-platforms-becoming-the-new-holy-grail-of-cyberattackers/

When Risks Become Reality

Below are real world examples of citizen development risks within organizations today.

Microsoft Power Apps Portals Data Leak (2021)

The Microsoft Power Apps Portals data leak was discovered by UpGuard in 2021, exposing a critical risk inherent to the citizen development model, specifically related to low-code/no-code platforms. The breach involved approximately 38 million sensitive records from applications built by dozens of companies, including American Airlines and U.S. government entities. The leaked data included Social Security numbers, employee IDs, and COVID-19 vaccination data. The core issue was not a software bug, but rather a systemic misconfiguration stemming from the platform's security defaults. Specifically, when a citizen developer enabled the OData API to expose data from a "list", the crucial privacy setting, "Table Permissions", was disabled by default. Because non-technical developers lacked the security expertise to switch this setting to "enabled manually", they inadvertently created thousands of web portals where sensitive backend data was anonymously accessible to anyone on the internet. This incident served as a stark example of how the speed and simplicity of low-code/no-code tools, combined with developers lacking security awareness, can turn an insecure default into an enterprise-wide, public exposure of sensitive data.

References:

- Lakshmanan, R. (2021, August 24). 38 million records exposed from Microsoft Power Apps of dozens of organisations. The Hacker News. https://thehackernews.com/2021/08/38-million-records-exposed-from.html
- UpGuard Team. (2021, August 23). By design: How default permissions on Microsoft Power Apps exposed millions. UpGuard. https://www.upguard.com/breaches/power-apps
- Zenity. (2021, November 18). The Microsoft Power Apps Portal Data Leak Revisited: Are you safe now?. Zenity. https://zenity.io/blog/research/the-microsoft-power-apps-portal-data-leak-revisited-are-you-safe-now

Systemic AI Code Vulnerability Reports (2025)

A 2025 GenAI Code Security Report by Veracode found that AI-assisted coding has not improved the security of generated software over time despite advancements in model size and functional accuracy. After analyzing over 100 Large Language Models (LLMS) across 80 coding tasks, the research revealed that LLM-generated code introduced a known security vulnerability in a significant 45% of cases. This consistent failure rate is systemic, stemming from the models' training on large, uncurated public code datasets containing numerous insecure examples. Coding flaws are then replicated by the LLMs when prioritizing functionality over security. Specifically, the models demonstrated extreme weakness in protecting against context-dependent flaws such as Cross-Site Scripting (XXS) and Log Injection.

References:

- Veracode. (2025). 2025 GenAI Code Security Report. Retrieved from https://www.veracode.com/resources/analyst-reports/2025-genai-code-security-report-2
- Cybernews. (2025, July 31). AI-generated code is functional, but not secure at all, researchers warn. Retrieved from https://cybernews.com/security/ai-generated-code-insecure-half-of-the-time/
- Endor Labs. (2025, August 12). The Most Common Security Vulnerabilities in AI-Generated Code. Retrieved from https://www.endorlabs.com/learn/the-most-common-security-vulnerabilities-in-ai-generated-code

Passion.io Breach (2025)

Passion.io, a popular no-code app-building platform, suffered a major data breach after a database was left publicly exposed without password protection or encryption. The exposed database contained over 3.6 million records, including users' names, email, addresses, payment and invoice details, profile images, creator-uploaded videos and PDFs, and internal business documents. This incident exemplifies the security risks of misconfigured infrastructure in no-code platforms, where users rely heavily on the providers' backend security but remain vulnerable to simple configuration errors.

References:

- Fadilpašić, S. (2025, June 5). More than 3 million records, 12TB of data exposed in major app builder breach. TechRadar. https://www.techradar.com/pro/security/more-than-3-million-records-12tb-of-data-exposed-in-major-app-builder-breach
- Fowler, J. (2025, June 4). Over 3 million records, including PII exposed in app-building platform data breach. vpnMentor. https://www.vpnmentor.com/news/report-passionapps-breach/

Acknowledgements

Contributors

Kayla Underkoffler

Shauna Rathbun

Reviewers

Michael Bargury

Ken Huang

Tatu Seppälä

Jon Gadsden

Sean Glencross