政策・技術論文 ｜ 各国政府関係者向け

既存資産を活かす国家サイバー耐性の再設計

PHP2Shell、ARPANET・UNIX・Machの歴史、メモリ安全性から導く二段階のDevSecOps政策

苫米地英人

カーネギーメロン大学CyLab, ジョージメイソン大学C5I研究所, コグニティブリサーチラボラトリィズ

2026年10月26日

目次

要旨
政策決定者のための要点
本稿の読み方
1　はじめに：問題の所在と本稿の主張
2　2026年10月に何が起きたか：PHP2Shellの事実関係
3　なぜ古い前提が今も危険なのか：メモリ安全性とC言語
4　歴史が示すもの：ARPANET、UNIX、BSD、Mach
5　PHPを使い続けながら守る：設計条件と緊急対応
6　サーバー構成と責任分担：VPSでもリスクの本質は変わらない
7　言語の選択肢：Common Lisp、Erlang、Rustが保証すること・しないこと
8　Common Lispによる代替実装と歴史的経験
9　隔離・再起動・サーバーレスの実際
10　DevSecOpsを業務とアーキテクチャにする
11　効果を測る：安全性と継続性を分ける
12　政府の義務化モデルと導入工程
13　AIを使った置換を安全に進める
14　結論：全面刷新を待たず、安全に変えられる能力を制度化する
付録A　防御コードの読み解き（実行確認済み）
付録B　既存のPHPを守る配備・運用テンプレート
付録C　政策担当者のための問答集
付録D　用語集
参考文献

要旨

2026年10月、PHP処理系の中核で発見された二つの欠陥（CVE-2026-109618、CVE-2026-109619、通称PHP2Shell）は、日本を含む多数の国で行政・金融・交通・報道・商取引のWebサービスを一斉に停止させた。日本では国の機関、自治体、銀行、航空、電力、大学、ホステ

ィング事業者を含む少なくとも246組織がサービス停止を告知し、欧州・カナダでも政府サイトが予防的に閉鎖された。欧州の政府機関への侵入事例も一時報告されたが、PHP2Shellによる初期侵入とは確認できないとして10月25日に撤回された。本稿が確認した公表情報の範囲では、PHP2Shellを初期侵入経路とする侵害は確認されておらず、被害の中心はサービスの停止である。[52,53,57,58]

本稿は、この事件を個別の不具合ではなく構造的な問題の現れとして扱う。PHPのアプリケーションはPHP言語で書かれるが、それを実行するエンジンはC言語で書かれている。C言語とUNIXは、参加者が限られ相互に信頼し合う研究共同体の時代に生まれ、メモリの確保・解放の正しさをプログラマに委ねる設計を今も受け継いでいる。今日の行政情報システムは、匿名かつ国家主体を含む攻撃者が常時接続してくる環境で、この基盤を大規模に使い続けている。

しかし、世界中のOSと言語処理系を一度に置き換えることはできない。本稿の中心的主張は、**OSやプログラミング言語を根本的に変えなくても、開発・調達・配備・運用の仕組みをDevSecOpsとして再設計すれば、攻撃にさらされる面、侵害の成立、侵害後の到達範囲、修正までの時間、復旧能力をそれぞれ縮小でき、それらの効果は掛け算で効く**という点にある。その上で第二段階として、外部入力を処理する高リスク部品から、ANSI Common Lisp、Erlang、Rustなどメモリ安全性と障害隔離に優れた言語・ランタイムへ、AIを補助に用いながら段階的に移行する。

本稿は、事件の事実関係を一次資料と修正コードの確認によって整理し、ARPANET・UNIX・BSD・Machの歴史から現在のリスクの由来を説明し、各国政府が義務化すべき統制と導入工程を提案する。また、サーバー構成別に影響と責任分担を整理し、同じ週に国内で起きたVPSへの不正アクセス事案（侵入経路は未公表）も踏まえて、VPSやクラウドという提供形態の違いがこの結論を変えないことを示す。技術者が検証できる厳密さを保ちつつ、政策担当者が判断に使えるよう、各節に平易な解説を付した。掲載コードは筆者の作成環境で実際にコンパイル・実行し、出力を併記した。

キーワード：サイバーレジリエンス、DevSecOps、メモリ安全性、PHP2Shell、ゼロトラスト、ソフトウェア供給網、Common Lisp、Erlang、Rust、政府調達

政策決定者のための要点

1. **今回の被害の中心は「停止」だった。**多くの組織は、自らのどのサーバーで、どの版のPHPが、どの設定で動いているかを即座に把握できず、修正を確実かつ迅速に配備する手段も持たなかっ

たため、サービス停止以外の選択肢を失った。これは技術の問題である以上に、管理と調達の問題である。

2. **根本原因は古い設計前提にある**。PHPエンジンはC言語で書かれ、メモリ管理の誤りが遠隔からのコード実行に直結し得る。C言語・UNIXは1970年前後、限られた研究者共同体のために設計された。米欧政府はすでに、メモリ安全でない言語からの計画的移行を求める方針を公表している。[39]

3. **全面刷新を待つ必要はない**。資産台帳、修正の自動配備、最小権限、ネットワーク分離、侵害検知、復旧訓練を組み合わせれば、既存のOSと言語のままでも被害の確率と規模を大きく下げられる。防火区画・スプリンクラー・定期点検・避難訓練によって古い建物を使い続けるのと同じ発想である。VPSやクラウドに移しても、この必要性は変わらない（第6節）。

4. **「大幅な改善」は測定で示す**。本稿は効果量を断定せず、公開資産の把握率、修正配備までの時間、越権アクセスの遮断率、復旧時間などの指標を義務化し、事前・事後で検証することを提案する。

5. **第二段階では高リスク部品から置き換える**。外部入力の解析、特権を持つ部品、ネイティブ拡張を優先し、Common Lisp、Erlang、Rustなどへ移行する。AIによる翻訳は有力な手段だが、人間の責任者、独立した試験と検証を必須とする。

6. **時間が最大の変数になった**。今回は公表前に悪用の試みが観測され、修正公開から数時間で数百万件の攻撃試行が記録された。欠陥の発見の一部にはAIが使われ、同じ能力は攻撃者も使い得る。修正を「数週間」で配備する体制では間に合わない。

本稿の読み方

本文は、事実（第2節）、原因（第3・4節）、対策の原理（第5〜9節）、対策の制度化（第10〜13節）の順に進む。各所に置いた**政策担当者向け解説**は専門用語を使わずに要点を説明し、**技術ノート**は専門家向けの補足である。付録Aは実行確認済みのコードとその読み解き、付録Bは既存のPHP環境に適用できる設定テンプレート、付録Cは政策担当者のための問答集、付録Dは用語集である。

事実の記述には出典番号を付した。2026年10月の事件に関する情報は日々更新されており、本稿は2026年10月26日（日本時間）までに確認できた資料に基づく。調達・配備の判断にあたっては、必ず最新の公式情報を再確認されたい。

1　はじめに：問題の所在と本稿の主張

1.1　設計前提と現在の脅威のずれ

1983年1月1日、ARPANETは通信規約をNCPからTCP/IPへ切り替えた。これは今日のインターネットへ至る大きな節目である。当時の参加ノードは米国内の選ばれた研究機関と、1973年に接続されたノルウェー（NORSAR）や英国（ロンドン大学UCL）など少数の海外拠点に限られていた。[10,63] UNIX（1969年〜）とC言語（1969〜73年、主に1972年に開発）も、ベル研究所の研究者が自らの研究環境のために作ったものである。[12,67]

この時代に安全性の意識が皆無だったわけではない。1973年のRFC 602は、弱いパスワード、遠隔からの不正利用、想定以上に広いアクセスという、今日にも通じる問題を既に指摘している。[11] しかし、不特定多数の匿名の攻撃者や、外国の国家主体が常時接続してくる環境を前提として、言語・OS・プロトコルを包括的に設計することは目標ではなかった。とりわけC言語は、性能と移植性のために、メモリの確保・解放とその範囲の正しさをプログラマの責任とした。この設計は、互換性を保つために現在まで受け継がれている。

2026年10月のPHP2Shellは、この「設計時の前提」と「現在の脅威」のずれが、社会全体の業務停止という形で顕在化した事例である。PHPは1995年に公開されたツールから発展したWeb時代の言語であり、それ自体はARPANET以前のものではない。[21] しかし、PHPを動かすエンジンはC言語で書かれ、C言語の性質を引き継いでいる。

1.2　本稿の主張と構成

本稿の主張は次の三点である。第一に、PHP2Shellの社会的被害の大部分は、欠陥そのものよりも、資産把握・修正配備・権限分離・復旧能力の不足によって拡大した。第二に、これらは既存のOSと言語を維持したまま、DevSecOpsという開発・運用の仕組みによって大幅に改善できる。第三に、それと並行して、外部入力を扱う高リスク部品から、メモリ安全な言語・ランタイムへの重点的な移行を進めるべきである。

本稿は、一次資料と筆者の研究開発経験に基づく政策・技術論文である。歴史・仕様・事件の事実には出典を示し、設計上の推論と政策提案には適用条件を明記する。修正内容については、PHPの公式ソースコードリポジトリに公開された修正コミットを直接確認した。[50]

1.3　証拠の区分

サイバー事件の報道では、性質の異なる事実がしばしば混同される。本稿では次の四つを区別する。政府の広報においても、それぞれについて何が確認されたのかを明示することが、過小評価と過剰反応の双方を避ける前提になる。

区分	意味	PHP2Shellでの例（10月26日時点）
脆弱性の存在	攻撃に悪用され得る欠陥がある	PHP 8.2〜8.5の修正前版、および保守終了版の多く[1,52]
攻撃の試行	攻撃が送られたことが観測された	公表前の10月17日にAmazonが観測、公表後に数百万件規模[5,52]
侵害の成立	攻撃が成功し、システム内で不正な処理が行われた	公表情報では未確認。欧州の政府機関の事例は、初期侵入経路を確認できないとして撤回（Unit 42、10月25日）[53]
予防的停止	侵害の有無とは別に、安全のため止めた	日本の少なくとも246組織、欧州・カナダの政府サイト[52,57,58]

侵害の公表がないことは安全の証明ではない。一方、停止したという事実だけから情報流出を推定してもならない。また、Unit 42が10月23日に報告し25日に撤回した欧州の政府機関の事例は、侵害の帰属、すなわちどの脆弱性が初期侵入に使われたかの確認がいかに難しいかを示している。脆弱なPHPアプリケーションの上で起きた攻撃であったことから当初はPHP2Shellの悪用と判断されたが、初期侵入との関係は確認できなかった。速報段階の報告は、確度と更新の有無を確かめたうえで政策判断に用いるべきである。[53]

1.4　脅威モデル

対象は、インターネットに公開されたサービス、内部API、開発・配備基盤、委託先との接続である。攻撃者には、外国の国家主体や犯罪組織に加え、国内で乗っ取られた端末、窃取された正規アカウント、悪意を持って改変された依存パッケージを含める。IPアドレスから推定される国は補助的な情報にすぎず、通信元の国籍や所在だけで信頼を判断することはできない。外部の攻撃者が内部の正規の権限を取得する経路を、設計時から評価する必要がある。これはゼロトラストの基本的な考え方である。[38]

2　2026年10月に何が起きたか：PHP2Shellの事実関係

2.1　PHPとは何か、なぜ影響が大きいのか

政策担当者向け解説

PHPは、Webサーバー上で動き、利用者の入力に応じてページを組み立てるためのプログラミング言語である。WordPress、Drupal、Joomla、Magentoなど、行政や企業のサイトで広く使われる仕組みの多くがPHPで動いている。[57] 利用者がフォームに入力したりファイルを送信したりすると、その内容はまずPHPのエンジンが受け取り、解析してからアプリケーションに渡す。今回の欠陥は、この「受け取って解析する」段階、つまりアプリケーションの作り方に関係なく必ず通る入口に存在した。したがって、各組織が自分のWebアプリケーションを丁寧に作っていても防げなかった。

W3Techsの調査では、サーバー側の言語が判明したWebサイトの69.2%がPHPを使用している（2026年10月25日時点）。[75] 攻撃面の規模については、計測方法の異なる複数の推計がある。Palo Alto NetworksのCortex Xpanseは、潜在的に脆弱な公開インスタンスを2,260万件超と推計した。[53] Modatの観測では、公表直前の10日間に4,384,302件のPHPサービスが1,886,612台のホスト上で確認され、版が読み取れるもののうち約79%が保守終了した系列であった。[55] Bitsightも、最も多く観測された系列が保守終了済みのPHP 7.4であり、版が判明した中で修正対象の8.2〜8.5系列は約3分の1にとどまると報告している。[62] いずれも脆弱性の有無を個別に確認した数ではないが、保守終了版の比率の高さは、正式な修正版へ単純に更新できない組織が多いことを意味する。

2.2　時系列

以下に、本稿の結論に関わる主要な事実を時系列で示す（時刻の表記がない項目は日付のみ）。

日時	出来事
2026年8月	Calif、OpenAI、Wizなど複数の研究者が、同じ根本原因に関わる欠陥を独立にPHP開発者へ報告。発見の一部ではAIを用いた解析が使われたと報告されている。[52,55]
10月17日	Amazonが自社のハニーポット（MadPot）で、公表前の悪用の試みを観測。[5]
10月18日	PHPの修正コミット「Avoid state corruption via bailout」が作成される（8.5系列で423ファイルを変更）。[50]
10月20〜21日	公表前に情報が漏えいし、誤ったCVSS値を伴う情報や攻撃コードが流通。Cloudflareは21日3時02分（UTC）にWAF（Webアプリケーションファイアウォール。表の下の解説を参照）の防御規則を有効化。[52,60]
10月21日 21時55分（JST）	高市首相がXで、今夜中の修正準備と、迅速な適用が難しい場合の機能の一時停止を呼びかけ。[56,61]
10月21日	ドイツ、フィンランド、ベルギー、オランダ（国立公衆衛生環境研究所RIVM）、フランス（県庁）、カナダの政府サイトが予防的に閉鎖。CERT-EUは更新できない場合の一時停止を勧告。[8,57,58]
10月21日 夜（JST）	PHPが修正版8.2.35、8.3.36、8.4.27、8.5.12を公開。保守終了の7.4、8.0、8.1にはソースの最小限のバックポートを提供。[1,54,62]
10月21〜22日	財務省がホームページの一部を一時停止し翌22日に再開。国税庁、厚生労働省もシステムを停止。IPAとJPCERT/CCが注意喚起を公表。[4,6,7,52]
10月22日	CloudflareのCEOが、公表前10時間で数千件、公表後45分で数万件、2時間以内に数十万件、投稿時点で数百万件の攻撃試行を観測したと公表。[52]
10月23日	PHPが、21日の修正で対処されない別経路（CVE-2026-115560）について、修正までセッションのアップロード進捗機能の無効化を案内。[1,54]

日時	出来事
10月23日	Palo Alto Networks Unit 42が、欧州の政府機関で侵入から2時間以内にパスワードハッシュと多要素認証の秘密情報が取得され、Active Directoryへの中継と管理者権限の奪取が試みられたと報告。攻撃者の操作にはAIエージェントの使用を示す特徴があったとした（25日に撤回）。[53]
10月25日	Unit 42が更新で、23日の追記を撤回。観測したのは脆弱なPHPアプリケーション上で行われた欧州の政府機関への攻撃であり、そのためPHP2Shellの悪用と判断したが、初期侵入が当該CVEによるものとは確認できないとした。[53]

政策担当者向け解説：WAF（Webアプリケーションファイアウォール）とは

WAF（Web Application Firewall）は、Webサーバーの手前に置かれ、届いた要求（HTTPの通信）の中身を検査して、攻撃とみられるものを遮断する装置やサービスである。通常のファイアウォールが「どこから、どの窓口へ」という通信の宛先で通すか止めるかを決めるのに対し、WAFは要求の内容、例えば送られてきたデータの形や大きさ、既知の攻撃に特有のパターンを見て判断する。Cloudflareのようなクラウド事業者が、利用者のサイトの前段でサービスとして提供する形態も多い。

WAFの利点は、ソフトウェア本体を直さなくても、検査の規則を追加するだけで短時間に防御を強められる点にある。今回もCloudflareは、修正版の公開の約9時間前からWAFの規則で攻撃を遮断していた。[60] 一方で限界もある。第一に、WAFを経由せずにサーバーへ届く通信（内部ネットワーク、別の入口、サーバーのアドレスへの直接の接続など）には効かない。第二に、規則は既知の攻撃の形に基づくため、攻撃者が要求の形を変えればすり抜けられ得る。今回も攻撃の変形が観測された。第三に、欠陥そのものはサーバー側に残る。したがってWAFは、修正までの時間を稼ぐ防御層の一つであり、修正の代わりにはならない。

2.3　二つの欠陥の仕組み

政策担当者向け解説：ホテルの鍵のたとえ

PHPのWebサーバーは、効率のために「ワーカー」と呼ぶ作業プロセスを使い回し、一つのワーカーが多数の利用者の要求を順番に処理する。ホテルが同じ部屋を次々に別の客へ貸すのに似ている。要求ごとの作業用メモリは、客の滞在中だけ使う部屋にあたる。

今回の欠陥の一つは、チェックアウト（要求の終了）の後も、館内の台帳に前の客の部屋番号が残ってしまう問題であった。部屋はすでに清掃され、次の客に割り当てられているのに、古い台帳を頼りに誰かがその部屋の扉を開けてしまう。攻撃者は、自分の要求でわざとこの状態を作り、次の要求でその「古い鍵」を使わせることで、サーバー上で任意の命令を実行できた。もう一つの欠陥は、メモリ不足で処理を急停止させたとき、部屋の準備が途中のまま非常口から飛び出してしまい、後片付けの担当者が中途半端な部屋を正常な部屋として扱ってしまう問題であった。

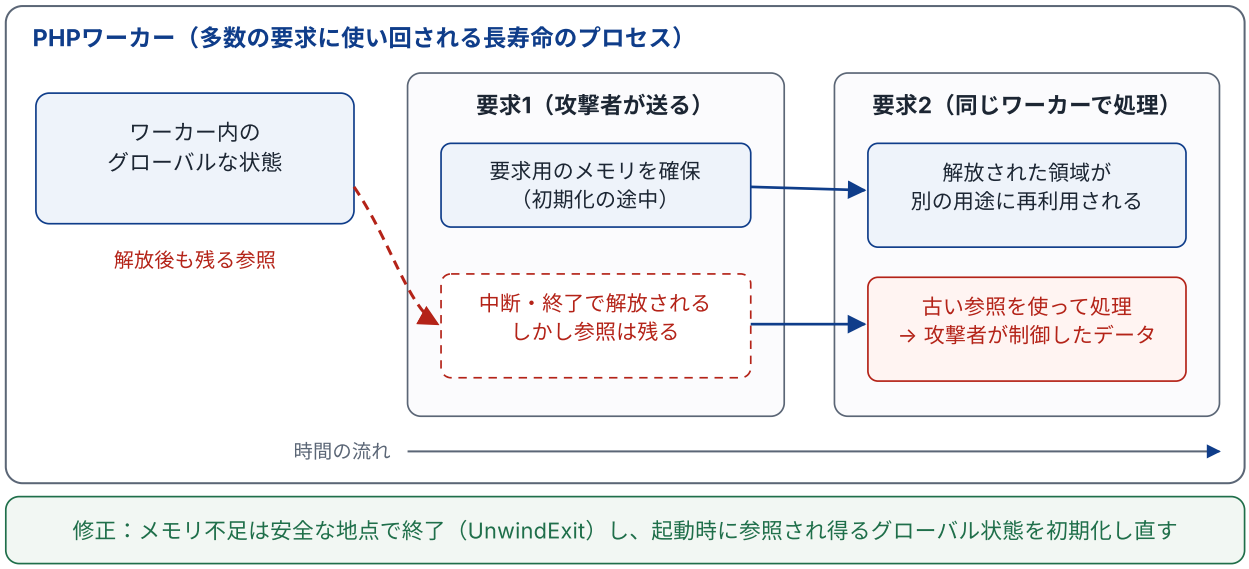

図1　ワーカーの再利用と「古い参照」：CVE-2026-109619の概念図（簡略化）

技術的には、二つの欠陥は次のように整理できる。いずれもメモリ安全性の欠陥である解放後使用（use-after-free、UAF）に分類される。[2,3,54]

技術ノート：解放後使用（UAF）とは何か

C言語のプログラムは、作業に必要なメモリ領域を確保し、使い終わると解放して返却する。解放後使用（UAF）とは、解放した後も残っている古いアドレス（ダングリングポインタ。宙に浮いた参照）を通じて、その領域を再び読み書きしてしまう誤りである。C言語では、メモリを解放してもポインタの値は自動的に無効化されず、その後の使用は言語規格上の未定義動作となる。

危険なのは、解放された領域がメモリ管理によって別の目的に再割り当てされ得る点にある。攻撃者が後続の入力でその領域を自分の用意したデータで埋められれば、プログラムは古い参照を通じて、そのデータを正規の内部状態として扱う。その中に関数ポインタや仮想関数表のように「次に実行する場所」を示す値が含まれていれば、制御の流れを奪われ、任意のコードを実行され得る。UAFが常に任意コード実行に至るわけではないが、これがUAFから遠隔での乗っ取りに至る典型的な経路である。ホテルのたとえでは、解放が「チェックアウト」、ダングリングポインタが「台帳に残った古い部屋番号」、再割り当てが「次の客への貸し出し」にあたる。CVE-2026-109619では、要求の終了時に解放されたメモリへの参照がワーカーのグローバル状態に残り、同じワーカーで処理される後続の要求がそれを使う。最小のコード例は第3節のコード1で示す。

欠陥	問題となる境界	重要な限定
CVE-2026-109618	メモリ上限超過（OOM）による処理の中断が、初期化途中の要求状態を残し、それを終了処理が扱ってしまう	mod_php（Apache組込み）では、アプリケーションのコードが動く前の経路が実証された。PHP-FPMでは起動時の失敗で子プロセスが終了するなど挙動が異なり、同じ経路を前提にしてはならない。
CVE-2026-109619	要求ごとのメモリの寿命と、ワーカー内に残るグローバルな参照の寿命が一致しない	同じワーカーの後続の要求が古い状態を利用する必要がある。古い状態を生む経路はOOMに限らない。

攻撃が成立した場合、コードはWebワーカーの権限で実行される。その実行アカウントは環境によって異なり、必ずしもwww-dataやrootではない。しかし、その権限で読めるデータ、データベースの接続情報、クラウドの認証情報が多ければ、最初の侵入だけで深刻な被害になり得る。さらに、そこを足掛かりに認証基盤やバックアップへ横展開されれば、被害は組織全体に及び得る。

2.4　修正の内容：「非常脱出」から「安全な地点での終了」へ

PHPのエンジン（Zend Engine）は、メモリ上限の超過などの致命的な事態に陥ったとき、C言語の`longjmp`という仕組みで、処理の途中から要求の最上位へ一気に制御を移していた（PHP内部では`zend_bailout()`と呼ばれる）。`longjmp`は途中の関数が行うはずだった後片付けを実行しない。[45] 筆者が確認した修正コミットは、その説明に「OOM時のbailoutをUnwindExitに置き換える」「起動時に参照され得る各種グローバル変数を適切に初期化し直す」と記している。[50]

修正後のメモリ管理は二段構えである。通常の上限（ソフトリミット）を超えた時点では、ただちに脱出せず「保留中のメモリ不足」を記録して処理を続け、仮想マシンの安全な地点で正規の終了手順（UnwindExit）に入る。その間にメモリ使用が上位の上限（ハードリミット）まで達した場合は、標準エラー出力に記録してプロセスを終了コード137で即時終了させる（`_exit(137)`）。ハードリミットは新しい設定項目`memory_hard_limit_percent`で調整でき、既定ではmemory_limitが1GiB以下なら200%、それを超える場合は1GiBの上乗せとなる。[50] この修正は8.5系列で423ファイル、約1万1,500行の追加に及び、多岐にわたるメモリ不足の経路について回帰試験が追加された。

> **技術ノート**
>
> `_exit(137)` は終了状態137でプロセスを終了させる呼び出しであり、シグナルSIGKILLを送る操作そのものではない（137はシェルがSIGKILLによる終了を表す慣用値と同じ数値である）。また、この修正は問題となった中断経路への対処であって、PHPの内部から `longjmp` 系の仕組みが全面的に除去されたことを意味しない。実際、修正後のコードにも `zend_try` ／ `zend_catch` による保護が残っている。[50]

なお、Linuxディストリビューションは、版番号を変えずに修正だけを取り込む（バックポートする）ことが多い。本稿の作成環境（Ubuntu 24.04 LTS）では、PHPの版表示は8.3.6であり、修正版8.3.36より古く見えるが、パッケージ8.3.6-0ubuntu0.24.04.13の変更履歴には10月19日付で本件の修正パッチが記録されていた。[51] 版番号だけで脆弱性の有無を判定すると、修正済みの環境を誤って停止させたり、逆に未修正の環境を見逃したりする。資産台帳には、パッケージの修正履歴まで記録する必要がある。

2.5　10月23日の追加情報

PHPは10月23日、セッションのシリアライズ方式を実行中に変更するという特殊な組み合わせの設定とアプリケーションにおいて、21日の修正では対処されない別の遠隔コード実行の経路（CVE-2026-115560）があるとし、修正版の公開まで`session.upload_progress.enabled=0`によってアップロード進捗機能を無効化するよう案内した。カナダのサイバーセキュリティセンターも同日、同じ緩和策を勧告している。[1,54] この機能は既定で有効であり、上記の作成環境でも有効であった。したがって、21日の更新を適用しただけで、すべての設定・経路について恒久的な安全が得られたとはいえない。

2.6　被害の実像：停止・試行・侵害を分けて見る

日本。セキュリティ研究者による集計では、10月下旬までに少なくとも246組織がサービスの停止を告知し、その大半は予防的な停止であった。国の機関（厚生労働省、財務省、国税庁）、自治体18（札幌市、横浜市、三重県、大阪府、大阪市など）、金融機関8、交通・インフラ11（航空会社、電力会社など）、大学9、放送2のほか、ゲーム、動画・電子書籍、電子商取引、ホスティング事業者が多数含まれる。[52] 大手ホスティング事業者は、旧版を含む全系列への修正適用やアップロード進捗機能の一律無効化を行い、管理画面などを一時停止した後に順次再開した。個々のサービス停止は数時間から十数時間に及んだ例がある。[52,61]

10月24日時点の公開情報では、日本国内で本件を原因とする侵害の成立は確認されていない。[52] なお、同じ週に国内では、GMOインターネットのVPSサービスで、顧客情報の持ち出しを伴う不正アクセスが確認された。侵入経路は公表されておらず、PHP2Shellとの関係は確認されていない（第6.6節）。[82,83]

海外。欧州ではドイツ、フィンランド、ベルギーが予防的にサイトやサービスを停止し、オランダのRIVMやフランスの県庁のサイトも閉鎖された。[57] カナダでも連邦政府の一部サイトが一時閉鎖された。[58] 侵害については、Unit 42が10月23日に欧州の政府機関での侵入後活動を報告したが、25日の更新で、PHP2Shellによる初期侵入とは確認できないとして撤回した。[53] 本稿が確認した公表情報の範囲では、PHP2Shellを初期侵入経路とする侵害は確認されていない。 また、北朝鮮系とみられる攻撃基盤が日本を含む12の国・地域の未修正ホストを探索していたとの分析もある（帰属の確度は中程度とされる）。[59]

> **政策担当者向け解説：「停止」も被害である**
>
> 情報が盗まれていなくても、行政手続、金融取引、医療・介護情報の提供、航空券の予約、電子書籍の閲覧が止まれば、国民生活と経済に損失が生じる。今回、多くの組織が停止を選んだのは、①自組織のどこでPHPがどの版で動いているかを即座に把握できない、②修正を数時間で安全に配備できない、③侵入されても被害が広がらないという確信が持てない、という三つの理由による。逆に言えば、この三点を平時から整えておけば、停止を最小限にしたまま守ることができる。これが本稿の提案の出発点である。

2.7　事件が示した五つの構造問題

1. **共通基盤の単一障害点**。世界中の無数のサービスが同じ処理系に依存しているため、一つの欠陥が国境と業種を越えて同時に影響する。
2. **資産把握の欠如**。委託先、パッケージ製品、管理ツール、機器に組み込まれたPHPまで把握していなければ、修正の要否すら判断できない。
3. **保守終了版の多さ**。観測された環境の多くが保守終了した系列であり、正式な修正版へ単純に更新できない。[55,62]
4. **時間の圧縮**。公表前の漏えいと悪用の試み、公表後数時間での大量の攻撃試行、AIを用いた欠陥の発見が重なった。修正の配備に許される時間は、日単位から時間単位へ縮んでいる。[5,52,55]
5. **境界防御の限界**。WAFは公表前から有効な防御層になったが、WAFを経由しない通信には効かず、攻撃の変形も観測された。WAFは修正の代わりにはならない。[60]

3　なぜ古い前提が今も危険なのか：メモリ安全性とC言語

3.1　メモリ安全性とは何か

政策担当者向け解説：借りた部屋は返したら使わない

プログラムは動作中、作業のためにコンピュータのメモリを「借り」、用が済んだら「返す」。メモリ安全性とは、借りた範囲の外を触らないこと、返した後の場所を使わないことが、言語や実行環境によって保証されている性質をいう。C言語ではこの保証がなく、プログラマが一行でも誤れば、他の目的に再利用された場所を読み書きしてしまう。攻撃者はこの誤りを突いて、そこに自分の命令を置き、プログラムに実行させる。

米国では2022年以降、NSA、ホワイトハウスの国家サイバー長官室（ONCD）、CISAが相次いでメモリ安全な言語への移行を促している。[68,69,70] 米国CISA、NSA、FBIと豪州、カナダ、英国、ニュージーランドの当局は、2023年12月の共同文書で、Microsoftの脆弱性の約70%、Google Chromiumの脆弱性の約70%がメモリ安全性の問題であったという報告を引き、ソフトウェア製造者にメモリ安全な言語への移行計画（ロードマップ）を求めた。[39] 米国CISAとFBIは、製造者が2025年末までにロードマップを公表すべきだとしている。[71]

次の二つのコードは、同じ誤りがC言語とRustでどう扱われるかを示す（付録A.1に詳細）。C言語のプログラムはそのままコンパイルでき、実行すると解放済みの領域を読む。これは言語規格上の未定義動作であり、何が起きても規格違反とはならない。検査用の計測機能（AddressSanitizer）を付けてコンパイルした場合に限り、実行時に誤りが報告される。Rustでは同じ誤りがコンパイルの段階で拒否され、実行可能ファイルが作られない。

コード1　C言語：解放後使用（UAF）の最小例　uaf.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *session = malloc(16);   // (1) 16バイトを借りる
    if (session == NULL) return 1;
    strcpy(session, "user=alice");
    free(session);                // (2) 借りたメモリを返す（解放）
    printf("%s\n", session);      // (3) 返した後に使う：解放後使用
    return 0;                     //     C言語では未定義動作になる
}
```

実行結果（gcc 13.3、-fsanitize=address、アドレスは省略）

```
==25626==ERROR: AddressSanitizer: heap-use-after-free on address 0x… at pc 0x… bp 0x… sp 0x…
READ of size 2 at 0x… thread T0
freed by thread T0 here:
previously allocated by thread T0 here:
```

コード2　Rust：同じ誤りはコンパイル時に拒否される moved.rs

```rust
fn main() {
    let session = String::from("user=alice"); // (1) 変数が所有者になる
    drop(session);                            // (2) 所有権を手放して解放
    println!("{}", session);                  // (3) 解放後の使用はコンパイル不可
}
```

コンパイル結果（rustc 1.97.0）

```
error[E0382]: borrow of moved value: `session`
 --> moved.rs:4:20
  |
2 |     let session = String::from("user=alice"); // (1) 変数が所有者になる
  |         ------- move occurs because `session` has type `String`, which does not implement
the `Copy` trait
3 |     drop(session);                            // (2) 所有権を手放して解放
  |          ------- value moved here
4 |     println!("{}", session);                  // (3) 解放後の使用はコンパイル不可
  |                     ^^^^^^^ value borrowed here after move
  |
help: consider cloning the value if the performance cost is acceptable
```

ここで重要なのは、Cでも検査ツールを使えば誤りの多くを発見できるという点である。ただし、それはツールを開発工程に組み込み、試験がその経路を実際に通った場合に限られる。DevSecOpsは、このような検査を「担当者の心がけ」から「毎回必ず通る工程」へ変える仕組みである（第10節）。

3.2　責任箇所を正しく分ける

Cは言語、GCCはコンパイラ群、glibcはCの標準ライブラリの実装、Zend EngineはPHPの実行エンジンである。これらを「GNU C」という一語にまとめると、どこに責任があるかを誤る。例えば2024年のCVE-2024-2961は、glibcのiconv関数がISO-2022-CN-EXTへの文字コード変換で出力バッファを最大4バイト超えて書き込む欠陥であり、2026年のPHPの要求状態管理の欠陥とは別物である。[9]

また、メモリ安全性は主として「一つのプロセスの中で、オブジェクトの範囲と寿命を守ってアクセスすること」に関わる。OSの仮想記憶による保護は、別のプロセスのメモリへのアクセスを防ぐ別の層の仕組みであり、同じプロセス内のUAFは防がない。PHPのスクリプトは通常、生のポインタを扱わないが、それでもエンジンやネイティブ拡張の欠陥には、外部からの入力で到達し得る。

3.3　longjmpと後片付けの保証

C言語の`setjmp`／`longjmp`は、保存した実行位置へ非局所的に制御を移す仕組みである。その使用自体が常に未定義動作になるわけではない。対応する`setjmp`の呼び出しがない場合、それを含む関数が既に終了している場合、別のスレッドで`setjmp`した場合などに未定義動作となる。[45] 問題は、`longjmp`が途中の関数の後片付けを一切実行しない点にある。GCC独自の`__attribute__((cleanup))`も、変数が通常の方法で有効範囲を出るとき（および例外の巻戻し時）に呼ばれる仕組みであり、`longjmp`による脱出では実行されない。[77] 後片付けを前提に書かれたコードほど、強制脱出の後に「中途半端な状態」を残しやすい。

なお、「GCCの最適化が一般に攻撃しやすいメモリ配置を保証する」という主張には根拠がない。本件の本質はGNU環境に固有の問題ではなく、強制脱出によって後片付けを飛ばす設計と、要求をまたいで残る参照という、C言語で書かれた長寿命プロセスに共通する構造の問題である。

言語ごとの「脱出時の後片付け」の保証範囲を比較すると、次のようになる。いずれの言語も、電源断やOSによる強制終了（SIGKILL）まで後片付けを保証するものではない。

仕組み	脱出時の後片付け	保証の範囲外
C：`longjmp`	実行しない（途中の関数を飛び越える）	後片付けはすべてプログラマの責任
C++：例外	巻戻し時にデストラクタを実行	`std::terminate`、`abort`、強制終了
Common Lisp：`unwind-protect`	正常終了と非局所脱出の双方で後片付けを実行[23]	処理系の破損、OSによる強制終了、規格外のスレッド強制停止
Rust：`Drop`	有効範囲の終了とpanic時の巻戻しで実行	`panic=abort`設定、プロセスの強制終了（付録A.6で実測）
Erlang：プロセス終了	プロセスのヒープを回収し、監視者に通知	VM全体のメモリ枯渇、ネイティブ関数（NIF）の障害[27]

4　歴史が示すもの：ARPANET、UNIX、BSD、Mach

> **政策担当者向け解説**
>
> 今日の情報システムの土台の多くは、1970〜80年代に、限られた研究者が互いを信頼し合う環境で作られた。当時の技術者が無能だったのではなく、設計の目的が違ったのである。土台は互換性を保ちながら半世紀にわたって改良されてきたが、根本の設計思想は簡単には変えられない。歴史から学ぶべきことは「昔は安全を考えていなかった」ではなく、「設計時の前提と現在の脅威にずれがあり、そのずれを運用と構造で埋める必要がある」ということである。

4.1　ARPANETからインターネットへ

RFC 801は、1983年1月1日までにNCPからIP/TCPへ完全に移行することを目標に掲げた。[10] この日は重要な節目であるが、相互接続ネットワークの研究と運用はそれ以前から進んでおり、「その日にARPANETが突然インターネットに変わった」という説明は単純にすぎる。1973年6月15日には、ノルウェーのNORSARが衛星回線を経由して米国外で最初のARPANETノードとなり、直後に英国UCLも接続された。[63] 初期のネットワークは参加者が限定されていたが、国外との接続も安全への問題意識も存在した。RFC 602は、容易に推測できるパスワード、遠隔からの不正利用への無防備、想定よりはるかに広い範囲からのアクセスを警告している。[11]

4.2　UNIXとC言語

Dennis Ritchieによれば、UNIXは1969年に生まれた。[12] C言語は1969〜73年に形作られ、最も創造的な時期は1972年であった。[67] UNIXは多人数で一台の計算機を共有するためのアクセス制御を備えていたが、それは今日のように不特定多数の敵対的な相手と常時接続する環境への包括的な防御とは異なる。C言語は、ハードウェアに近い効率と移植性を優先し、メモリの範囲と寿命の正しさをプログラマに委ねた。この選択が、半世紀後の今もOS、言語処理系、ネットワーク機器の大部分を支えている。

観点	設計当時（1970年代）	現在
利用者	互いに顔の見える研究者・技術者	匿名の不特定多数、自動化された攻撃基盤
攻撃者の想定	内部の誤用、いたずら、パスワード漏えい	国家主体、犯罪組織、AIを用いた自動攻撃
接続	専用線・限られた拠点	常時接続、クラウド、委託先、IoT
修正の配布	テープや個別の連絡	公表から数時間で攻撃が始まる中での即時配備
メモリ管理	性能のためプログラマに委ねる	同じ設計が互換性のため継承され、主要な脆弱性の原因に

4.3　BSDとネットワークの普及

BSDは1970年代後半にUNIXへの追加ソフトウェアとして発展し、1983年8月に公開された4.2BSDはTCP/IPとソケットAPIの普及に大きく貢献した。[14,79] ただし、これはTCP/IPの発明やすべての最初の実装と同じではない。また、BSDがTCP/IPを採用した目的は研究用ネットワークへの接続であり、敵対的な環境を前提とする安全設計が主眼ではなかった。viやcshはBSDの初期の成果であるが、OpenSSHはOpenBSDの開発者が1999年12月に公開したものであり、年代が異なる。[16] 初期のAT&T由来コードに必要だったライセンスと、後年の再配布可能なBSDコードのライセンスも区別すべきである。[15]

4.4　商用UNIXとLinux

AT&T UNIX全体をSVR4（System V Release 4）と定義するのは正確でない。SVR4は1989年に出荷された、System V、BSD、Xenixを統合する後期の版である。1993年のUSL（UNIX System Laboratories）のNovellへの売却が、初めて他社にUNIXを使わせた出来事でもない。商用UNIXはそれ以前から各社にライセンスされており、Sunは競争相手であると同時に統合の協力者でもあった。[13] IBMは2000年12月に、翌年に約10億ドルをLinuxに投じると発表し、企業におけるLinux利用の大きな転機となった。[17] しかし、市場での普及は安全性の証明ではない。今日の行政システムの多くはLinuxの上で、C言語で書かれた多数の部品を使って動いている。

4.5　1988年の教訓：Morrisワーム

1988年11月、Morrisワームはインターネット上のUNIX系計算機に広がった。Spaffordの分析によれば、ワームはfingerdの入力処理におけるバッファあふれ（境界を確認しない`gets`の使用）、sendmailのデバッグ機能、計算機間の信頼関係とパスワードの推測を組み合わせて侵入した。[64] C言語のメモリ安全性の欠如が、ネットワークを通じた大規模な被害に直結し得ることは、38年前から知られていたのである。その後の対策は、個々の欠陥の修正、緩和技術、運用上の防御を積み重ねるものであった。PHP2Shellも、同じ系譜の上にある。

4.6　Mach：互換性を保ちながら内部の境界を作り直す

カーネギーメロン大学（CMU）のMachプロジェクトは1985年に始まり、1994年まで続いた。[18] 当時の4.2BSD系UNIXのカーネルは、当初の想定にない機能の追加によって見通しが悪くなっており、Machはプロセス間通信（IPC）、仮想記憶、軽量スレッド、多数のプロセッサへの対応、移植性、そしてUNIXとの互換性を目標とした。初期にはBSDのコードを取り込んで実装されたが、後にUNIXの機能をカーネルの外のアプリケーションとして動かす構成へ進んだ。CMUの研究者は、1989年3月からこの分離された環境を学内で運用していたと記している。[19] Machの系譜は、NeXTSTEPやOSF/1などを経て広がった。

ポート（通信の出入口）を介したIPCは、「どの主体に、何を依頼する権限を渡すか」を明示的に設計することを可能にする。カーネルを小さくすることは特権を持つコードの範囲を絞る機会になるが、各サーバーの権限、メッセージの検証、ドライバ、共有資源の扱いを誤れば脆弱になる。「分割した」ことと「安全な境界を実証した」ことは別である。

AppleのXNUは、CMUのMachカーネルにFreeBSD由来の部品と、ドライバを書くためのC++ API（IOKit）を組み合わせたハイブリッドカーネルである。[20] macOSやiOSにMachの系譜が残ることは、純粋なマイクロカーネル構成がそのまま使われていることを意味しない。なお、Mach-Oは実行ファイルの形式であり、カーネルそのものではない。

> **政策上の教訓：**Machは、外部に見せる互換性（UNIXのAPI）を保ったまま、内部の責任の境界を作り直せることを示した。行政システムにおいても、全面刷新を待たずに、既存のアプリケーションを動かしたまま権限・通信・障害の境界を再設計できる。これがDevSecOpsによる第一段階の考え

> 方である。

5　PHPを使い続けながら守る：設計条件と緊急対応

PHPは、1995年6月にRasmus Lerdorfが公開した「Personal Home Page Tools」から発展した。[21] PHPそのものの歴史と、現在のエンジン・拡張機能・アプリケーションのリスクは分けて評価すべきである。同じPHPアプリケーションでも、修正済みの処理系、不要な拡張機能の削除、最小権限、データベースへのアクセスの限定、秘密情報（パスワードや鍵）の短命化によって、侵害時の損失は大きく変わる。

5.1　緊急対応の順序

1. **把握：**公開資産、PHPの実行形態（mod_php、PHP-FPM、CGI等）、拡張機能、パッケージの修正状況（版番号ではなく修正履歴）を確認する。委託先、パッケージ製品、管理ツール、機器に同梱されたPHPを含める。
2. **修正：**公式の修正版、またはディストリビューションのバックポートを適用する。CMSやプラグインの更新は、PHP処理系の更新の代わりにならない。保守終了版は、供給者の修正か、保守対象の版への移行を要する。
3. **再起動：**すべてのワーカーを再起動し、古い状態を持つプロセスを残さない。
4. **確認：**実際に動いている構成（版、設定、`session.upload_progress.enabled`）を確認する。CLIとWebサーバーの設定は異なる場合がある。
5. **調査：**公表前後からのログを保全し、不審なプロセス、ファイル、外部通信、アカウントを調べる。修正は既に成立した侵入を取り消さない。[4,54]

5.2　暫定策の限界

`memory_limit=-1`（上限なし）は、公式のアドバイザリが特定の条件で説明した暫定策であり、万人向けの安全な設定ではない。上限を外せば、メモリ枯渇によるサービス停止や、基礎的な確保の失敗という別の経路が残るため、修正版の適用を優先する。[2] 入力サイズの上限も、処理中のデータ構造の膨張率を直接制限するとは限らない。nginxの`client_max_body_size`や`client_body_buffer_size`は要求本文の受け入れに関する設定であり、PHP内部のメモリ使用量の上限ではない。[47] WAFは公表前から有効な追加の防御層となったが、WAFを経由しない経路には効かない。[60] 暫定策は時間を稼ぐ手段であり、修正の代わりにはならない。

政策担当者向け解説：止めずに守れた組織の条件

修正版の公開から数時間のうちに、検証済みの更新を全サーバーへ配り、全ワーカーを入れ替え、稼働中の設定を確認できる組織は、サービスを長時間止める必要がない。これを可能にするのは個人の技量ではなく、資産台帳、自動化された構築・試験・配備の工程、段階的な切り替えの手順、そしてそれを平時から訓練していることである。第10節で、その仕組みを具体的に述べる。

6　サーバー構成と責任分担：VPSでもリスクの本質は変わらない

PHP2Shellの影響は、OSの種類やサーバーの提供形態の名前では決まらない。Linux、Windows Server、VPS、専用サーバー、共有ホスティング、コンテナのいずれであっても、未修正のPHPがWeb要求を処理していれば対策の対象となり得る。PHPの公式告知も、Web要求を処理する各種の実行方式を広く更新の対象としている。[1] 一方で、すべてのOSと実行方式で同じ攻撃が実証されたわけではない。第1.3節の証拠の区別と同じく、「更新の対象」「条件付きの影響」「実証済みの攻撃経路」を分けて判断し、一般的な資料だけで個別の環境の安全を認定しないことが重要である。

6.1　四つの軸で判断する

1. **OS：**Linux、Windows Server、BSD系など。OSの名前だけでPHPの状態は決まらない。
2. **PHPの実行方式（SAPI）：**Apacheへの組込み（mod_php）、PHP-FPM、IISのFastCGI、CGI、FrankenPHP、LiteSpeedなど。
3. **配置形態：**物理サーバー、VPSや仮想マシン、コンテナ、共有ホスティング、PaaS、サーバーレス。
4. **運用責任：**誰がPHPを更新し、全ワーカーの入替えと稼働状態の確認までを行うか。

これらは排他的な分類ではない。例えば「Windows ServerのVPS上でIISのFastCGIを使い、保守は自社で行う」という組合せがあり得る。四つの軸のどれか一つ、例えば「VPSだから」「Windowsだから」という理由だけで、対策の要否を決めてはならない。

政策担当者向け解説：VPSは「賃貸マンションの一室」である

VPS（仮想専用サーバー）は、一台の大きな物理サーバーを仮想化の技術で区切り、利用者ごとに専用のサーバーのように貸し出す提供形態である。賃貸マンションにたとえれば、仮想化は住戸と住戸の間の壁にあたる。壁が厚ければ、隣の住戸の火事や泥棒は自分の住戸へ及びにくい。

しかし、壁は自分の住戸の玄関の鍵の掛け忘れを補わない。VPSの中で動くOSやPHPに欠陥があれば、攻撃者は壁を越える必要もなく、正面の玄関から入ってくる。さらにマンションには、全住戸に通じる管理事務所、合鍵の保管庫、共用の設備がある。VPSでいえば、事業者の管理画面、サーバーの作成や復元の仕組み、遠隔操作や監視の基盤である。ここが破られれば、壁の厚さに関係なく多数の住戸が同時に危険にさらされる。したがって、VPSに移したことは安全の理由にならない。住戸の中（利用者による更新と権限の管理）と管理事務所（事業者の管理基盤）の双方で、第10節で述べるDevSecOpsの仕組みが必要になる。

6.2　構成別の判断一覧

次の表は、公式の更新対象と各基盤の仕様を組み合わせた運用上の整理である。特定の事業者や製品について攻撃の成立を実証した一覧ではない。

構成	影響の考え方	主な確認・対処
Linux＋Apache（mod_php）	最優先で更新する。アプリケーションのコードが動く前の攻撃経路が実証されている。[2]	実際に読み込まれているPHPモジュール、配布元の修正、Apacheワーカーの入替え
Linux＋nginxまたはApache＋PHP-FPM	FPM固有の挙動はあるが、安全とは断定できない。[2,3]	FPMの版、pool、拡張、起動時の設定、全子プロセスの入替え
Windows Server＋IIS（FastCGI）	未修正のPHPは更新の対象。Linuxでの実証結果をWindowsの全構成へ一般化しない。	ハンドラーが指すphp-cgi.exe、適用されるphp.ini、アプリケーションプールと実行ID[86]
Windows＋Apacheや同梱型の環境	PHPを同梱する製品も対象。Windows Updateでは別途導入したPHPは更新されない。	接続方式、同梱のPHP、拡張DLL、配布パッケージ、古いプロセス
自己管理のVPS・クラウドの仮想マシン	PHPは仮想化の内側で動く。VPS内部のデータを盗むのに、仮想化の境界を越える必要はない。	ゲストOSとPHPの更新担当、実行権限、DBやバックアップへの到達範囲[87]
専用の物理サーバー	他の契約者との同居を避けても、PHP自体の欠陥は残る。	同じ機械上の複数サービス、共有の秘密情報、管理権限の分離

構成	影響の考え方	主な確認・対処
共有ホスティング・管理付きVPS	契約上の保守範囲が決め手。「管理付き」という名称だけでは判断できない。	事業者によるPHPの修正と全プロセス更新の証拠、利用者側のCMSやプラグインの更新
Docker・Kubernetes	コンテナ内のPHPも対象。再起動だけでは同じ脆弱なイメージが再び使われる。	修正済みイメージの再構築、全レプリカの更新、稼働中のダイジェストの確認（第9節）
PaaS・サーバーレス	事業者管理のランタイムか、利用者が持ち込むランタイムかで責任が変わる。要求ごとに完全に初期化されるとは限らない。[32]	実際のPHPの供給元、カスタムイメージやレイヤー、配備版と再利用の挙動
BSD系・その他のOS	OSの名前で免疫を仮定しない。悪用の可否はビルドと構成に依存する。	パッケージ、ports、独自ビルドの修正。jailはホストのカーネルを共有するため、jail内部も別に更新する[88]
PHPを使わない部分	静的サイトやASP.NETなど、PHPを実行しない部分は直接の対象ではない。	同居する管理画面、旧サイト、バックエンドにPHPが残っていないか

6.3　Linuxサーバー：稼働中のPHPを特定する

nginxやApacheは要求の入口であり、その名前だけではPHPの実行方式は決まらない。nginxからPHP-FPMへ渡す構成、ApacheにPHPを組み込む構成（mod_php）、ApacheからPHP-FPMへ渡す構成を区別する。mod_phpでは要求の解析段階で問題に到達するため、CMSの更新やログイン認証だけでは守れない。[2] PHP-FPMは、要求の起動に失敗した子プロセスを終了させる点でmod_phpと挙動が異なるが、正常に起動した後の処理など別の条件もあり、更新を省略してよい理由にはならない。[2,3]

確認では、配布元の修正情報（第2.4節のとおり、版番号ではなく修正履歴）に加え、ソースからの独自導入、別のリポジトリ、管理パネル配下の複数のPHPまで台帳に含める。そのうえで、更新が実行中のプロセスに反映されたかを確かめる。付録B.7のコード16は、この確認を`/proc`の情報から機械的に行う。サイトごとに実行ユーザーとDBの権限を分け、秘密情報と書込み可能な領域を限定することも欠かせない。rootを奪われなくても、Webワーカーの権限で読める情報は盗まれ得る。

技術ノート：「更新した」と「更新後の実体で動いている」は別である

Linuxのパッケージ更新は、通常、古いファイルを削除して新しいファイルを置く。既に動いているプロセスは、削除された古い実行ファイルやライブラリをメモリに読み込んだまま実行を続けるため、`/proc/<PID>/exe` のリンク先や `/proc/<PID>/maps` には「(deleted)」と表示される。mod_phpではPHP本体がApacheに読み込まれた共有ライブラリ（libphp）であり、PHP-FPMではマスタープロセスと各poolの子プロセスがPHPを実行する。いずれも、全プロセスの入替えを確認するまで修正は完了していない。一方、侵害が疑われる場合は、再起動の前に、調査に必要な証拠（メモリ、ログ、不審なファイル）を保全するかを判断する（第5.1節）。

6.4　Windows Server：IISとPHPの更新を分ける

PHP2ShellはPHP処理系（Zend Engine）の内部の問題であり、Linuxのglibcだけに依存する欠陥ではない。[50]Windows Serverでも、使用しているPHPの修正状態を確認する。IISのFastCGIでは、ハンドラーマッピングが要求を処理するphp-cgi.exeを指定し、FastCGIの定義がそのプロセスの引数や環境変数を決める。[86] 新しい版を別のフォルダーに展開しただけでは切り替わらない。PHPの配布版（x64かx86か、スレッドセーフ版か否か）と拡張DLLの組合せ、実際に適用されるphp.ini、FastCGIの引数と環境変数を確認し、更新後には古いphp-cgi.exeのプロセスが残っていないことを確かめる。アプリケーションプールを分離し、実行IDとNTFSの権限を絞る。付録B.8のコード17は、IISの設定ファイルから実際のPHPを特定し、版とハッシュを記録する。

技術ノート：10月26日のWindows版の再ビルドと「版番号の罠」

PHPは10月26日、Windows版の配布パッケージを8.2.35-1、8.3.36-1、8.4.27-1、8.5.12-1として再ビルドした。これは同梱する圧縮ライブラリBrotliの欠陥（CVE-2026-111497、伸張処理における境界外書込み）を修正した1.2.1への更新であり、PHP2Shellの修正とは別件である。この欠陥には、cURL拡張でHTTP応答の自動伸張を有効にしている場合に到達し得る。cURLの既定では無効だが、アプリケーションやHTTPクライアントのライブラリが有効にする場合がある。[84,85] PHPのソースと実行時の版は変わらないため、`php -v` の表示は再ビルドの前後で同じである。版の表示だけでは配布物の中身を識別できないという点で、第2.4節のLinuxのバックポートと同じ構造の問題である。配布物の取得元、ビルド番号、SHA-256ハッシュを、部品表（SBOM）と資産台帳で管理する理由がここにある。

6.5　VPS・専用サーバー・共有ホスティングの責任分担

VPSは配置・提供の形態であり、OSでもPHPの修正の仕組みでもない。仮想マシン型のVPSではゲストOSごとに境界があるが、OSレベルの仮想化を使うサービスではカーネルを共有する場合がある。どの分離方式でも、内部のPHPと内部のデータの間には別にアクセス制御が必要である。Webワーカーの権限でコードを

実行された場合、その権限で読める設定ファイル、DBの認証情報、APIの鍵、アップロード領域が被害の範囲となる。rootやAdministratorの奪取、物理ホストへの脱出が成立しなくても、重大な事故になる。逆に、PHPの侵害が自動的にホスト全体の完全な支配を意味するわけでもない。

提供形態	更新の担当	利用者が確認する証拠
自己管理のVPS・IaaS	通常は利用者がゲストOSと導入したPHPを管理する。実際の契約が優先する。[87]	修正パッケージ、適用日時、再起動後に稼働している版
管理付きVPS・専用サーバー	事業者の保守範囲にPHP、拡張、管理パネル配下の版が含まれるかを契約で確認する。	対象の一覧、適用状況、未対応の例外、担当者と期限
共有ホスティング	PHPは事業者、アプリケーションは利用者という分担が多い。契約で確定する。	選択中のPHPの修正、ワーカーの更新、旧版の扱い。CMSの更新だけでは完了しない

VPSのメモリを増やしても根本的な修正にはならない。逆に、メモリの少ない環境で`memory_limit`を無制限にすると、別のサービス停止の危険を高める（第5.2節）。要求単位の上限、同時実行数、ホスト全体の容量は別々に設計する。

6.6　日本のGMO事案が示すこと

PHP2Shellと同じ週に国内で大きく報じられたのが、GMOインターネットが提供するFX自動売買向けのVPS「お名前.com デスクトップクラウド」への不正アクセスである。公表と報道によれば、経過は次のとおりである。

日付	経過
10月19日	利用者からの問い合わせを受け、調査を開始。[83]
10月20日	一部の利用者のサーバー環境に、第三者が不正なプログラムを設置していたことを確認。新規申込みとコントロールパネルを停止。[81,83]
10月21日	午前4時から全プランのサーバーを順次停止。影響の可能性があるサーバーは2万6,739件（被害が確定した件数ではない）。[81,83]
10月22日	取引口座のログイン情報等が閲覧され、不正に利用された可能性を公表。金融庁は金融先物取引業協会を通じてFX事業者に通知し、利用者への注意喚起と取引の監視を求めた。[82,83]
10月23日	利用者に全セッションの切断を要請（パスワードを変更しても既存のセッションが残り得るため）。利用者自身の端末のバックアップとウイルススキャンを、予防的措置として要請。[82,83]

日付	経過
10月24日	顧客の情報が外部に持ち出された事実を確認したと公表（種類と件数は未公表）。攻撃者とみられる者から金銭の要求を受けたとし、応じない方針を表明。不正取引による損失の補償と全利用者への見舞金を発表。現行の基盤をそのまま再開せず、利用者には同意を得たうえで独立したクラウド基盤の代替サービスへの移行を案内。本件を能動的サイバー防御の「発動第1号」事案と位置付け、政府の関係機関に攻撃基盤の追跡・遮断・無力化を要請。[82,83,89]
10月25〜26日	利用者の端末を預かり、ホワイトハッカーのチームが解析。利用者に端末のフルスキャンとバックアップを改めて要請。攻撃者から、10月27日12時に顧客情報の公開とランサムウェア攻撃を行うとの予告を受けたと公表。[90]

侵入の経路は公表されていない。GMOは「本サービスのシステムにセキュリティ上の問題が存在」したとしているが、PHPが稼働していたか、PHP2Shellが関係したかは明らかにされていない。[89,90] 不正なプログラムの確認は10月20日であり、PHP2Shellの修正版の公開（21日夜）より前、公表前の悪用の試みが観測された17日より後であるが、時期が近いことだけでは両者の関係は推定できない。本稿は本件をPHP2Shellの被害として数えない。また、攻撃者側の主張（侵入や情報窃取の経路など）は検証されていない。

それでも、本件は本稿の主張にとって重要である。第一に、VPSという提供形態でも、被害は利用者のサーバーの内側に及んだ。不正なプログラムは利用者のサーバー環境に置かれ、サーバー内に保管された取引口座の認証情報が狙われた。仮想化の境界は契約者どうしを分けても、各サーバーの内側で動くソフトウェアと、そこに置かれた秘密情報を守るものではない。さらに事業者は利用者の端末を預かって解析しており、懸念は利用者自身の端末にも及んでいる。[90] 第二に、事業者は被害の拡大を防ぐため、全プランのサーバーを停止した（影響の可能性があるとされたのは2万6,739件）。多数のサーバーに通じる管理画面、サーバーの作成・復元、遠隔接続の基盤は、一点が破られれば全体に波及し得る共通の管理基盤である。本件でどの部分が破られたかは公表されていないが、全体の停止を選ばざるを得なかったことは、この危険を示している。第三に、事業者は既存の基盤をそのまま再開しない方針を示し、利用者には独立したクラウド基盤上の代替サービスへの移行を案内した。[89] 侵害された環境を使い続けず、信頼できる状態から作り直すという点で、第10節の不変インフラの考え方に通じる。

したがって、PHPが稼働していたか否かにかかわらず、この事案が組織に突き付けた問いはPHP2Shellと同じである。どこで何が動いているか（資産台帳）。誰がいつ修正し、それが稼働中の実体に反映されたか（更新の責任と証拠）。サーバーにどの秘密情報が置かれ、侵害時にどこまで到達されるか（最小権限と秘密情報の管理）。認証情報とセッションを直ちに失効できるか（対応）。信頼できる成果物から作り直せるか（復旧）。VPSへの移行は、これらの問いを事業者と利用者の間で分担し直すだけであり、問いそのものを消すものではない。DevSecOpsの体制は、自前のサーバーでも、VPSでも、クラウドでも必須である。

技術ノート：VPSの三つの層と、それぞれの防御

VPSの安全は三つの層に分けて考える。(1) ゲストの内側：OS、PHP等の処理系、アプリケーション、保存された秘密情報。ここは利用者（管理付きの場合は契約に応じて事業者）の更新と最小権限の責任である。(2) 分離の層：ハイパーバイザーやコンテナのランタイム。契約者どうしを分けるが、ゲスト内部の欠陥は防がない。OSレベルの仮想化ではカーネルも共有する。(3) 管理の層：コントロールパネル、サーバーの作成・複製・復元、遠隔デスクトップやコンソールの中継、監視やバックアップのエージェント。多数のゲストに特権でアクセスできるため、職務の分離、多要素認証、短命な資格情報、変更の記録と監視が必要である。

侵害対応では、パスワードの変更だけでは足りない場合がある。GMO事案でも、取引口座のパスワードを変更しても既存のログインセッションが残り得るとして、全セッションの切断が求められた。[83] 対応手順には、資格情報の変更に加えて、セッションとトークンの失効を含める必要がある。

政策担当者向け解説：「VPSだから」「大手だから」は安全の理由にならない

GMOの事案とPHP2Shellは同じ週に起きたため混同されやすいが、公表された事実の範囲では、両者の関係は確認されていない。政策の議論では、まず両者を区別する必要がある。

そのうえで、両者に共通する教訓ははっきりしている。サーバーをVPSやクラウドで借りても、中で動くソフトウェアの更新、保存する秘密情報の管理、侵害時の封じ込めと作り直しは、誰かが責任を持って実行しなければならない。行政機関がVPSやクラウドを調達する際は、事業者の規模や名前ではなく、次の四点を契約で求め、定期的に確認すべきである。①更新の担当範囲と期限、②更新が稼働中の環境に反映された証拠、③管理基盤の保護（多要素認証、権限の分離、監視）、④侵害時の通知、セッションの失効、再構築の手順と訓練の記録。また、能動的サイバー防御による攻撃基盤への対処は重要な新しい手段であるが、被害を受けた組織自身の封じ込めと復旧の能力を代替するものではない。

6.7　その他の構成で注意すること

構成	注意点
コンテナ・Kubernetes	一般的なLinuxコンテナはホストのカーネルを共有する。更新の対象はホストOSだけではなく、PHPを含むイメージを修正して再配備し、全レプリカの実行イメージを確認する。自動再起動や定期交換はパッチではない。永続ボリューム、DB、窃取された資格情報は再生成では直らない（第9節）。
PaaS・サーバーレス	事業者管理のランタイムと、利用者が持ち込むPHPの実行ファイル、コンテナ、レイヤーを分けて台帳に載せる。実行環境やグローバル状態、`/tmp`は再利用され得る。[32] 実行時間を短くしても、その要求中に読める秘密情報やDBの権限の危険は残る。

構成	注意点
FrankenPHP・LiteSpeed・CGI・CLI	PHPを組み込む他の実装も更新の対象として確認する。mod_phpやFPMの説明を全実装にそのまま当てはめない。要求ごとに終了するCGIと複数の要求を扱うFastCGIでは、要求間に状態が残る条件が異なるが、片方の経路が成立しないことからPHP全体の安全を推論しない。CLIだけのバッチ処理は公開HTTPサーバーと同じ露出ではないが、外部からの入力や長時間動く独自のワーカーがあれば評価する。
CDN・WAF・ロードバランサーの背後	入口での遮断やサイズの制限は補助策である。オリジンサーバーへの直接の接続経路も閉じる。複数のバックエンドがあれば全台を確認する。認証の前にPHPが要求を解析する構成では、アプリケーションのログイン認証だけでは守れない。[2]

6.8　管理者・委託先へ確認する項目

次の記録がそろえば、「更新したつもり」と「稼働中の構成に反映された状態」を区別できる。行政機関が委託先に提出を求める項目としても使える。

1. **公開範囲：**対象のドメイン、管理画面、API、オリジンへの直接の経路、旧サイト
2. **OSと配置：**OS、物理・VPS・コンテナの別、仮想化の方式、保守の担当
3. **実行の実体：**PHPの実行ファイルまたはモジュール、SAPI、版、拡張、実行ID
4. **修正の証拠：**配布元のアドバイザリ、パッケージの版、バックポートの記録、イメージのダイジェスト、配布物のハッシュ
5. **設定の反映：**実際の起動時設定とサイト固有の上書き、アップロード進捗機能、容量と並列数
6. **プロセスの反映：**全ワーカー、全pool、全レプリカが入れ替わった記録
7. **動作の確認：**ログイン、アップロード、API、セッション、決済などの主要業務
8. **被害の範囲：**そのPHPから読める秘密情報、DBの権限、書込み可能な領域、バックアップの操作権限
9. **侵害の調査：**関連するログと変更履歴の保全、不審なプロセス・ファイル・通信の調査
10. **復旧：**侵害が疑われる場合の隔離、資格情報とセッションの失効、信頼できる成果物による再構築、データの整合性の確認

修正の適用は既知の侵入経路を閉じる作業であり、過去に侵害がなかったことを証明する作業ではない。証拠を失うだけの再起動を避け、緊急の封じ込めと必要な証拠の保全を両立させる。問うべきは、サーバー構成の名前ではなく、修正済みのPHPが実際に動いているか、侵害時の権限が限定されているか、更新と復旧を繰り返せるかである。これが第9節の隔離と第10節のDevSecOpsの前提となる。

7　言語の選択肢：Common Lisp、Erlang、Rustが保証すること・しな

いこと

第二段階の移行先として、本稿はANSI Common Lisp、Erlang、Rustを有力な候補とする。三者は異なる方法で、C言語に由来する欠陥の一群を構造的に減らす。ただし、特定の言語を採用したこと自体を安全性の証明とはしない。

選択肢	有益な仕組み	保証しないこと
ANSI Common Lisp	ガベージコレクションで管理されたオブジェクト、条件システム（condition／restart）、`unwind-protect`による脱出時の後片付け、稼働中の再定義	無限のメモリ、OSによる強制終了時の後片付け、FFI（他言語呼出し）や処理系自体の無欠陥、取引の自動的な整合性
Erlang／OTP	独立したヒープを持つ軽量プロセス、メッセージ通信、監視と監督（supervision）による障害の局所化と回復、稼働中のコード入替え	VM全体のメモリ枯渇やネイティブ関数（NIF）の障害からの隔離、決まった時間内の再起動、外部への副作用の自動取消し
Rust	所有権・借用・型による、安全なコード内でのメモリ範囲と寿命の静的な保証、`Drop`による後片付け	`unsafe`やFFI部分の自動検証、メモリリーク・デッドロック・認可の誤りの排除、`panic=abort`時の後片付け

Common Lisp。`handler-bind`のハンドラは、スタックを巻き戻さずに、エラーが通知された現場の文脈のまま実行される。ハンドラは状況を調べ、下位の関数が用意した「リスタート」（回復の選択肢）を選んで呼び出すことができる。リスタートへの制御移動の際には、その地点まで適切な巻戻しが行われ、途中の`unwind-protect`の後片付けが実行される。[23,24,25] つまり、「どう回復するか」という方針を上位が決め、「どこで回復するか」を下位の現場が定義するという分業が言語として可能になる（付録A.4）。ただし、`unwind-protect`が保証するのは言語が扱う正常終了と非局所脱出であり、電源断、SIGKILL、処理系の破損、規格外のスレッド強制停止を越える保証ではない。後片付け自体も失敗し得る。

Erlang。Erlangのプロセスは、OSのプロセスではなく、同じ仮想マシン（BEAM）の中で動く軽量プロセスである。各プロセスは独立したヒープを持ち、他のプロセスのメモリを直接参照できない。障害を起こしたプロセスだけを終了させ、監督者が新しいプロセスを起動するという「Let it crash」の設計は、純粋なErlangコードの範囲では強力な障害の局所化を実現する。[26] ただし、ネイティブ関数（NIF）が異常終了すればVM全体が停止する。[27] また、監督者による再起動は、再起動の頻度の制限、依存関係の順序、二重処理を防ぐ設計と組み合わせる必要がある。

Rust。Rustの安全性の保証は、`unsafe`の内部やコンパイラ、ライブラリが必要な約束を守ることを前提とする。[28] panicの既定の動作では巻戻しが行われ、途中の値の`Drop`が実行されるが、`panic=abort`の設定、メモリ確保の失敗による終了、OSによる強制終了では巻戻しを期待できない。[29] 本稿ではこの違いを実際に確認した（付録A.6）。ガベージコレクションがないことは、実行時の処理が皆無であることも意味しない。

Common LispからErlang、Rust、さらにKubernetesへと、一本の直接的な系譜があると断定することはできない。本稿では、障害の局所化、資源の所有、回復という共通の設計課題に対する異なる解として、これらを比較する。

8　Common Lispによる代替実装と歴史的経験

8.1　PHPをマネージドな実行環境の上に実装するという選択

PHPの処理系を、C言語のような手動のメモリ管理ではなく、ANSI Common Lispのような強力なマネージド実行環境の上に構築していれば、今回のような欠陥の一群は実装の段階で避けられた可能性が高い。Common Lispでは、メモリはガベージコレクションによって管理され、プログラムは生のポインタを扱わない。解放済みの領域を指す参照という状態そのものが、安全な言語の範囲では作れない。処理の中断は条件システムと`unwind-protect`によって扱われ、途中の後片付けが言語として保証される。

この方向には歴史的な裏付けもある。CMUCL（CMU Common Lisp）のマニュアルは、その開発がカーネギーメロン大学の「Research on Parallel Computing」契約のもとでDARPA（米国防高等研究計画局）の資金を受けたと記録しており、ネイティブコードのコンパイラ、ガベージコレクタ、FFIを備えていた。[46] この来歴は技術選択の重要な文脈であるが、軍事予算で開発されたことそのものは安全性の証明ではない。

本稿の主張は、合理的な設計仮説として次のように表現される。PHP相当の言語処理系をCommon Lispで構築し、安全なオブジェクト操作と明示的な状態遷移に限定するなら、手動の解放と生ポインタの操作に由来する欠陥の一群を、実装の水準で避ける余地がある。ただし、ANSI規格は無限のヒープや要求ごとの資源の隔離を規定しない。Cで書かれたライブラリをFFIで同じプロセスに組み込めば、その危険も持ち込まれる。

次のコードは、Common Lispが「途中で失敗しても後片付けを必ず行う」ことを示す最小の例である。解析の途中でエラーが起きても、一時バッファは必ず解放され、確保中のバッファ数は0に戻る。

コード3　Common Lisp：unwind-protectによる後片付けの保証　`l1_unwind.lisp`

```lisp
;;; 後始末を言語仕様として保証する unwind-protect（教材用）
(defvar *open-buffers* 0)                ; 確保中の一時バッファ数（観測用）

(defun allocate-temporary-buffer ()
  (incf *open-buffers*)
  (make-array 16 :element-type '(unsigned-byte 8)))

(defun free-temporary-buffer (buffer)
  (declare (ignore buffer))
  (decf *open-buffers*))

(defun parse-multipart-data (request buffer)
  (declare (ignore buffer))
  (when (eq request :malformed)          ; 解析途中の失敗を模擬
    (error "budget exceeded while parsing"))
  :parsed)

(defun process-web-request (request)
  (let ((buffer (allocate-temporary-buffer)))
    (unwind-protect
         (parse-multipart-data request buffer)  ; 保護される本体
      (free-temporary-buffer buffer))))         ; 正常終了でも脱出でも実行

(print (list (process-web-request :ok) *open-buffers*))
(print (list (handler-case (process-web-request :malformed)
               (error () :rejected))            ; エラーは上位で要求拒否に変換
             *open-buffers*))
```

実行結果（GNU CLISP 2.49.95+、-ansi）

```
(:PARSED 0)
(:REJECTED 0)
```

政策担当者向け解説

C言語の `longjmp` は、火災のとき片付けをせずに非常口から飛び出すようなものである。Common Lispの `unwind-protect` は、どの出口から出る場合でも、出口の手前で必ず片付け担当者が仕事をする建物にあたる。片付けが言語の規則として保証されるため、「中途半端に壊れた部屋」が次の利用者に渡ることがない。標準のファイル操作（`with-open-file`）もこの仕組みで、途中で何が起きてもファイルを閉じる（付録A.3）。

8.2　なぜPHPはCで実装され続けたのか

それでも、PHPがC言語のZend Engineで実装されてきたことには、歴史的・実用的な理由がある。第一に、PHPが普及した1990年代後半から2000年代初頭には、Common Lispの実行環境はメモリ消費が大きく、安価な共有ホスティングで多数の要求を処理するには不利であった。要求ごとにプロセスを起動するCGI方式では、起動の速い小さなCのプログラムが有利だった。第二に、PHPの強みは、MySQL、OpenSSL、libxml2など、Cで書かれた膨大な既存ライブラリを拡張機能としてそのまま取り込めることにあった。第三に、Zend Engineは、要求の終了時にメモリを一括で捨てる、短命な処理に特化した高速なメモリ管理をCで実装していた。そして皮肉にも、その単純さと、要求をまたいで生き続けるワーカーとの組み合わせが、今回の欠陥の温床となった。

比較にあたっては、PHPとの互換性、拡張機能の代替、ガベージコレクションによる停止時間、同時処理性能、運用要員、検証費用を含めるべきである。PeachPieのように.NET上でPHPを動かす実装は代替ランタイムの実例であるが、互換性の評価とネイティブ依存の点検が不可欠であり、あらゆる遠隔コード実行から自由であるとはいえない。[44] 言語を置き換えなくても、危険なネイティブ処理を別のプロセスへ移すという中間策もある。

筆者の政府研究開発：LispacheとBullet Proof Server

筆者は1998〜2003年に、通商産業省の政府予算による研究開発において、Franz社のAllegro Common Lispを用い、Lispache、Bullet Proof Serverと呼ぶWebサーバー、メールサーバー、DNSサーバー等を開発した。この経験は、本稿で論じる「言語処理系の資源管理と、システム全体の耐障害性を一体として設計する」という考え方の実践的な背景である。

LispacheはApache上位互換のWebサーバーである。通常のWebサーバーでは、例えば `https://crl.co.jp/CyberAcademy/` というURLの `CyberAcademy/` 以降の「/」による区切りが、UNIXの `public_html` 以下のディレクトリ構造をそのまま外部に示している。HTTPには、このようにファイルの内部構造を外部に見せてしまうリスクがある。Lispacheはこれを低減するため、`CyberAcademy/…/…/` といったディレクトリ構造を実際には持たず、`index.html` などのファイルそのものも存在させなかった。各要素はすべて暗号化されたオブジェクトデータベースにオブジェクトとして格納され、GETでアクセスされた時点で、あたかもディレクトリやファイルがあるかのようにリアルタイムに生成された。したがって、UNIXのルート権限が奪われても、そこにあるのは暗号化された巨大なオブジェクト格納ファイルだけであり、内容を直接改竄したり盗んだりすることはできない。

Bullet Proof Serverは、同様の機能を他のWebサービスにも拡大した。デザイナーの制作物やデータ等を格納するサーバーは物理的にネットワークから切り離し、更新時のみ、インターネット側のサーバーへ別経路で暗号化して転送した。さらに、インターネット側のサーバーとコンテンツ提供サー

バーの内容の同一性を、別のモニターサーバーが監視した。また、Lispの動的な性質により、保守やバージョンアップは完全に無停止で行えた。システムは稼働開始から約10年後、ビルの建て替えでやむなく停止するまで、一度も止まらずに稼働した。

CRLの公式の政府プロジェクト一覧には、通商産業省の「使用による課金を実現する流通・供給・再利用の高度化環境の開発」（MMCA-2005）や「次世代動的分散共有VM型ホームサーバシステムの開発」（11情技応1317号）などが記載されている。英語版でも、次世代動的オブジェクト指向セキュア計算アーキテクチャ、コンテンツ流通環境、分散共有動的VM型ホームサーバーの研究開発が紹介されている。[48,49]

これらの経験から本稿が重視するのは、メモリ管理、異常時の処理、サービス間の責任の分界を、設計の段階から組み合わせることである。今日のDevSecOps政策では、この設計原則を、継続的な検証、権限の制御、更新と復旧の運用へ接続する。

8.3　NASA Remote Agent：言語・検証・診断の組合せ

NASAの探査機ディープ・スペース1号では、1999年5月17日の週に、自律制御ソフトウェアRemote Agentの飛行実験が行われ、JPLは6月7日に全目標の達成を発表した。[31] その実行系（Executive）は多スレッドのLispで書かれていた。飛行中、ある事象を待つ処理が進まなくなる停止が発生した。NASAの事後分析によれば、原因は条件待ちの周囲に排他制御（クリティカルセクション）が欠けていたことによる二つのスレッド間の競合状態であり、それは1997年の飛行前のモデル検査（SPIN）で別の部分系に見つかった誤りと同型であった。誤りは、飛行前の検証の対象外だった兄弟の部分系に再び持ち込まれていた。[30] 開発者の回想によれば、地上のチームは探査機からバックトレースを取得して原因を突き止め、待たれていた事象を手動で発生させて処理を再開させた。原因は、独自に設計した言語の安全保証を迂回する低水準の関数が呼ばれていたことにあった。[65,66]

この事例から引き出すべき教訓は、「Lispだから事故が起きない」ではない。安全な抽象化（言語）、形式的な検証、稼働中のシステムの診断可能性、地上の運用能力を組み合わせることで、高信頼のシステムを支えられること。そして、安全な抽象化を迂回する低水準の呼出しが、検証の外側で危険を持ち込むことである。後者は、Rustの`unsafe`やあらゆる言語のFFIに対しても同じく当てはまる。

9　　隔離・再起動・サーバーレスの実際

コンテナは通常、ホストのカーネルを共有する。名前空間（namespaces）、資源制御（cgroups）、非rootでの実行、特権機能（capabilities）の削減、システムコールの制限（seccomp）、強制アクセス制御（MAC）を組み合わせて、侵害の波及を制約する。機密度が高い場合や、攻撃者と同じ計算機に同居する可

能性がある場合は、仮想マシン、マイクロVM、専用ホストを選ぶ。なお、通常のOSのプロセスにも仮想記憶による分離は存在する。「コンテナ以前は、一つのCプログラムが落ちれば必ずサーバー全体が落ちた」という説明は誤りである。[34,35,36]

Kubernetesの再起動と再配置は可用性のための仕組みであり、侵害の完全な除去ではない。同じ脆弱なイメージを再配備すれば再び侵入される。窃取された資格情報、改変されたデータベース、永続ボリューム、侵害されたノードや構築基盤（CI）は、別途の対処を要する。復旧時間は、監視の周期、再起動の抑制（バックオフ）、スケジューリング、起動処理に依存し、一律に1秒以内といった保証はない。

AWSの公式文書は、Lambdaの実行環境が再利用されること、関数の外側で宣言したオブジェクトや`/tmp`の内容が複数回の呼出しをまたいで残り得ることを明記している。[32] したがって、「サーバーレスでは毎回すべてのメモリが消えるのでUAFは不可能」とはいえない。FirecrackerはRustとKVMを用いた小さな仮想マシンモニタで攻撃面を抑えているが、ホストのカーネル、`unsafe`部分、設定と運用を含めた検証を要する。[33] いずれの仕組みも、最初の要求の処理中に完了した情報窃取を取り消すことはできない。

層	役割	主な手段	単独では防げないこと
コード	欠陥を作らない	メモリ安全な言語、レビュー、静的解析	設計・認可の誤り、`unsafe`／FFI
ランタイム	壊れた状態を次へ持ち越さない	`unwind-protect`、`Drop`、プロセス隔離、ワーカーの定期的な入替え	処理中に成立した窃取、ランタイム自体の欠陥
実行基盤	被害を区画に閉じ込める	コンテナの権限削減、マイクロVM、ネットワーク分離	共有カーネルの欠陥、過大な権限設定
運用	早く直し、早く気づき、確実に戻す	資産台帳、自動配備、検知、証拠保全、復旧訓練	台帳にない資産、訓練していない手順

10　DevSecOpsを業務とアーキテクチャにする

10.1　DevSecOpsとは何か

DevSecOpsは、開発（Development）、セキュリティ（Security）、運用（Operations）を、同じ変更管理と証拠の連鎖で結ぶ仕組みである。セキュリティを「リリース直前の検査」から「開発から運用までのすべての工程に組み込まれた、自動で繰り返される活動」へ変える。CIは継続的インテグレーション（変更を頻繁に統合し、そのたびに自動で構築・試験すること）、CDは継続的デリバリーまたはデプロイメント（検証済みの成果物をいつでも安全に配備できる状態に保つこと）を指す。NISTの安全なソフトウェア開発フレームワーク（SSDF、SP 800-218）は、既存の開発工程に安全な実務を組み込む枠組みを提供しており、2025年12月には改訂案（v1.2）が公表されている。[37,80] DevSecOpsは製品名でも検査ツールの集合でもなく、責任と証拠を伴う継続的な業務方式である。

政策担当者向け解説：古い建物を安全に使い続ける方法

古い建物をすべて建て替えることはできない。それでも人命を守れるのは、建築基準と消防の仕組みがあるからである。DevSecOpsは、ソフトウェアに対する同じ仕組みと考えることができる。

建物の安全	情報システムでの対応（DevSecOps）
建物の台帳と図面	資産台帳とソフトウェア部品表（SBOM）：どこで何が動いているか
建築確認・竣工検査	自動化された構築・試験・検査の関門（CI/CD）
防火区画・防火扉	最小権限とネットワーク分離：侵入されても燃え広がらない
火災報知器・スプリンクラー	振る舞いの検知と自動的な封じ込め
定期点検と改修	脆弱性情報との照合と、修正の迅速な配備
避難訓練	侵害を想定した演習と復旧訓練
消防法と査察	政府による義務化、調達条件、独立した評価

10.2　政府が導入すべき閉ループ

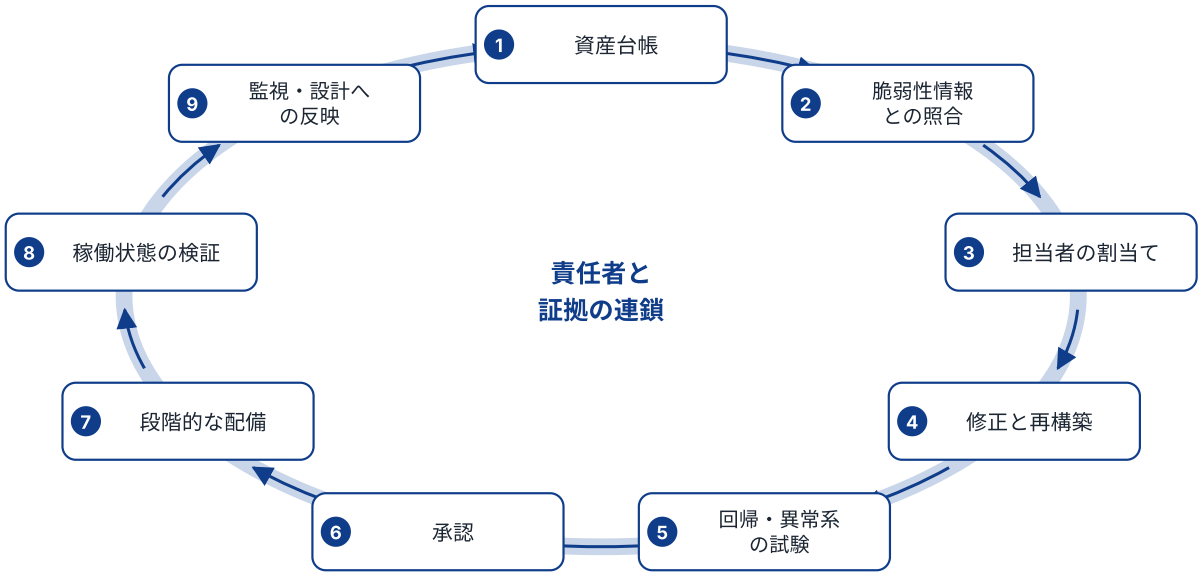

図2　政府が導入すべきDevSecOpsの閉ループ

政府が導入すべき閉ループは、資産台帳 → 脆弱性情報との照合 → 担当者の割当て → 修正と再構築 → 回帰試験と異常系の試験 → 承認 → 段階的な配備 → 稼働状態の検証 → 監視結果の設計への反映、という循環である。脆弱性情報が公表されただけで、安全な移行が自動的に完成するわけではない。各段階に責任者と記録を置き、どこで止まっているかを可視化することが重要である。

構成をコードとして管理し、承認済みの成果物をダイジェスト（内容から計算される一意の識別値）で固定する。稼働中のサーバーに手作業でパッチを当てるのではなく、修正を取り込んだ成果物を毎回清浄な状態から構築し、検証したものと同一の実体を本番へ配備する（不変インフラ）。ソフトウェア部品表（SBOM）とビルドの来歴を保存し、供給者・依存物・設定の変更を追跡する。電子署名は「誰が何を出したか」を確認する証拠であり、署名されたコードの安全性までは証明しない。構築基盤そのものも、最小権限、短命な資格情報、独立した承認で保護する。

ネットワーク上の位置だけに信頼を置かず、サービスごとに認証・認可を行う。公開Webサービスの権限から、バックアップの削除、データベース全体の抽出、クラウドの管理機能、認証基盤へ直接到達できないようにする。[38] Webサーバーを足掛かりに認証基盤やバックアップへ横展開する手口は、侵入後の典型的な攻撃手順である。製品を入れ替える前に、この権限の連鎖を断つことが、最も即効性のある改革である。

10.3　工程と対策の対応

工程	目的	主な実務
計画・実装	欠陥を作らない（シフトレフト）	脅威モデル、メモリ安全な言語の採用、レビュー、静的解析（SAST）、サニタイザ
構築・試験	既知の危険を本番の手前で止める	清浄な基盤からの再構築、依存物とイメージの検査、ファジング、異常系の試験、来歴の記録
配備・稼働	未知の欠陥が突かれても被害を区画に閉じ込める	最小権限、ネットワーク分離、段階的な配備、ワーカーとコンテナの入替え
監視・対応	早く気づき、証拠を残し、確実に戻す	可観測性、振る舞いの検知、証拠保全、資格情報の失効、復旧訓練、設計への反映

10.4　PHP2Shellの日に何が違ったか（例示）

以下は、本稿で述べた能力の有無によって、同じ事件への対応がどう変わるかを示す**例示的なシナリオ**である。特定の組織の実測ではない。

経過	統制が未整備の組織	DevSecOpsが整備された組織
公表の予告	PHPを使うシステムの洗い出しを始める。委託先への照会に時間がかかる	資産台帳から対象のシステム、版、修正履歴、責任者を即座に抽出

経過	統制が未整備の組織	DevSecOpsが整備された組織
修正版の公開	修正の適用手順が手作業で、試験環境もないため、サービスを停止して作業	修正を取り込んだイメージを自動で構築・検査し、一部の利用者から段階的に配備
数時間後	一部のサーバーが未修正のまま。どこまで直ったか分からない	全ワーカーの入替えと稼働中の版・設定を確認。例外は期限付きで記録
侵入されていた場合	Webサーバーの権限から認証基盤やバックアップへ到達される	分離と最小権限により被害が区画内に留まり、検知から封じ込めと資格情報の失効へ
翌日以降	停止の長期化、説明責任の困難	記録に基づく報告、再発防止の設計変更

11　効果を測る：安全性と継続性を分ける

11.1　被害は掛け算で決まる

評価の単位を、例えば「公開Webワーカーの侵害が、機密データベースの不正な更新に至る事象」と定める。この特定の経路について、Eを危険な入力への露出、Xを侵害の成立、Lを追加の重要資産への到達とすると、次の分解が成り立つ。

$$P(E \cap X \cap L) = P(E) \times P(X \mid E) \times P(L \mid E \cap X)$$

この式は事象の独立性を仮定していない（条件付き確率の連鎖である）。入力の制約や不要機能の停止は露出を、迅速な修正や安全な実装は侵害の成立を、最小権限と分離は後段への到達を、それぞれ抑える。この式は説明のための分解であり、すべての攻撃経路を網羅する保証式ではない。また、最初に侵害されたワーカー自体が持つ情報の価値も別に評価する必要がある。

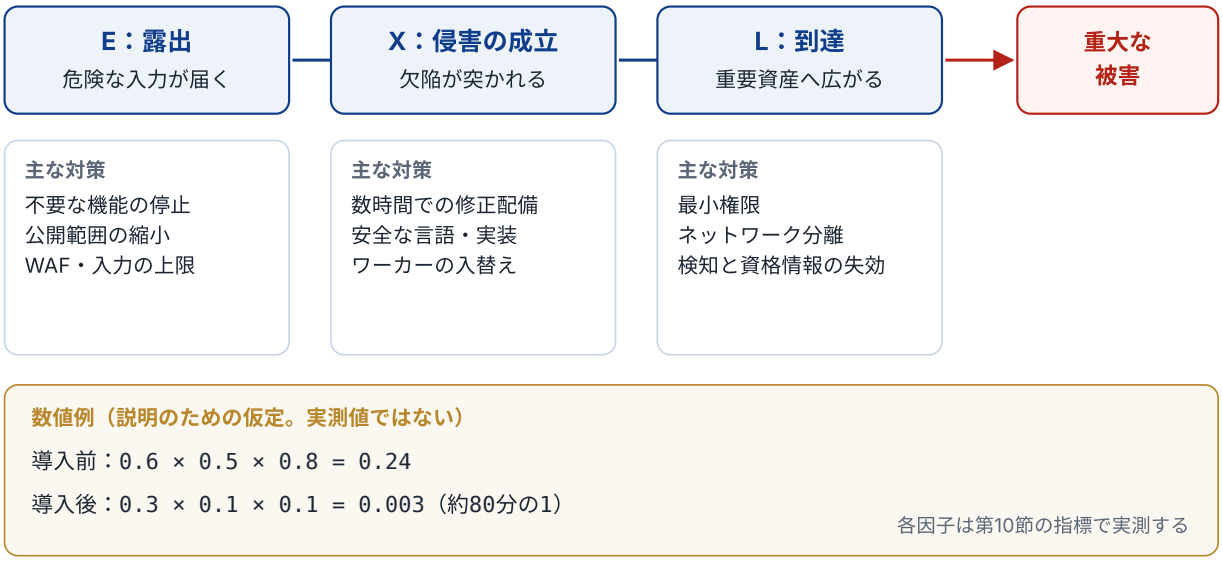

図3　攻撃の連鎖と、各段階を抑える対策

政策担当者向け解説：数値例（仮定）

以下の数値は仕組みを説明するための**仮定**であり、実測値ではない。ある公開サービスで、攻撃を受け得る機能が公開されている確率を0.6、公表後に修正が1週間遅れて攻撃が成功する確率を0.5、侵入後に認証基盤やデータベースへ到達できる確率を0.8とすると、全体は0.6×0.5×0.8＝0.24となる。DevSecOpsにより、不要機能の停止で露出を0.3に、数時間での修正配備で成功確率を0.1に、最小権限と分離で到達確率を0.1に下げられたとすれば、全体は0.3×0.1×0.1＝0.003となり、約80分の1になる。一つひとつの対策は完全でなくても、異なる段階に働く層を重ねることで全体の危険は大きく下がる。どの層がどれだけ効いているかは、次の指標で実測する。

11.2　測るべき指標

指標	定義	対応する因子
公開資産の把握率	外部からの観測で見つかった公開資産のうち、台帳に責任者付きで登録されている割合	E
保守終了資産の数	保守期間が終了した処理系・OS・部品を含むシステムの数と、その移行期限	X
修正配備時間	重大な脆弱性の公表から、全対象への配備と稼働状態の検証が完了するまでの時間（中央値と95パーセンタイル）	X
越権アクセスの遮断率	侵害を想定した試験で、Webワーカーの権限から重要資産への到達を遮断できた割合	L

指標	定義	対応する因子
検知・封じ込め時間	侵害の兆候の発生から検知、隔離、資格情報の失効までの時間	L
復旧試験の成功率	計画した復旧訓練のうち、目標時間（RTO）と許容データ損失（RPO）を満たした割合	継続性
変更失敗率	配備した変更のうち、障害や切戻しを要したものの割合	継続性

目標値は業務の重要度ごとに定める。比較は同じサービス群について、導入の前後で負荷、公開範囲、変更頻度を記録して行う。疑似攻撃を遮断できた比率を、実際の侵害確率と同一視してはならない。検査ツールの指摘件数の減少だけでは、検査対象の縮小や例外登録による見かけの改善を除外できない。復旧訓練では、バックアップからの復元だけでなく、鍵の失効、再発防止、データの整合性まで検証する。

DevSecOpsの能力と運用成績の関係については、DORA（DevOps Research and Assessment）の継続的な調査がある。2021年の報告では、最上位の群は変更のリードタイムと障害からの復旧時間がいずれも1時間未満であったとされる。[76] これは自己申告に基づく相関の調査であり、セキュリティ上の効果を直接証明するものではないが、「修正を数時間で安全に配備する」能力が現実の組織で達成されていることを示す。

12　政府の義務化モデルと導入工程

以下は、各国の行政命令、調達条件、監督制度に適合させるための**政策提案**であり、既に各国で施行されている共通の法令ではない。義務化の対象は、特定のクラウドや特定の言語ではなく、継続的に実証できる統制と責任とする。

1. **資産と責任：**稼働中のOS、言語処理系、拡張機能、依存物、公開経路と責任者を登録し、委託先にも提出を義務づける。版番号だけでなく、パッケージの修正履歴を記録する。
2. **更新可能性：**保守期限、緊急連絡、修正の提供、更新の試験、再起動・配備の能力を契約条件とする。
3. **変更の証拠：**レビュー、再現可能な構築の目標、部品表、来歴、署名の検証、稼働中の成果物のダイジェストを記録する。
4. **権限と隔離：**Web、データベース、管理機能、バックアップを分離し、許可する通信を限定する。例外には責任者と失効日を付ける。
5. **継続的な検証：**静的・動的解析、依存物の検査、ファジング、認可と異常系の試験を、リスクに応じて組み合わせる。
6. **事故対応：**監視、証拠保全、封じ込め、資格情報の更新、復旧、対外報告の役割と判断基準を定め、訓練する。

7. **データ保護：**ログの目的、保存期間、アクセス権を限定し、秘密情報や個人情報の無差別な収集を避ける。
8. **言語の移行：**危険な入力処理、特権を持つ部品、ネイティブ依存を優先したメモリ安全性のロードマップを要求する。[39,71]
9. **AIが生成したコード：**人間の責任者、独立したレビュー、隔離された試験、ライセンスと秘密情報の審査を必須とする。
10. **費用と監督：**中小の供給者向けの共通試験設備、基盤となるオープンソースの保守費、独立した評価、制度の定期的な見直しを予算化する。

12.1　導入の工程

期間（案）	優先する行動	成果の確認
直ちに〜30日	PHP等の公開資産の点検、修正の適用、旧ワーカーの排除、危険な公開機能の制限	稼働状態と責任者が判明し、緊急の例外に期限がある
1〜3か月	保護されたCI/CD、最小権限、監視と復旧訓練を代表的なサービスで実装	侵害想定試験と復旧試験の記録がある
3〜12か月	政府横断の再利用可能な基盤、委託契約の更新、高リスク部品の置換開始	修正配備時間と到達可能な権限が測定上改善する
12か月以降	検証済みの移行の拡大、技術的負債と供給者集中の再評価	継続性と安全性の改善を独立に評価する

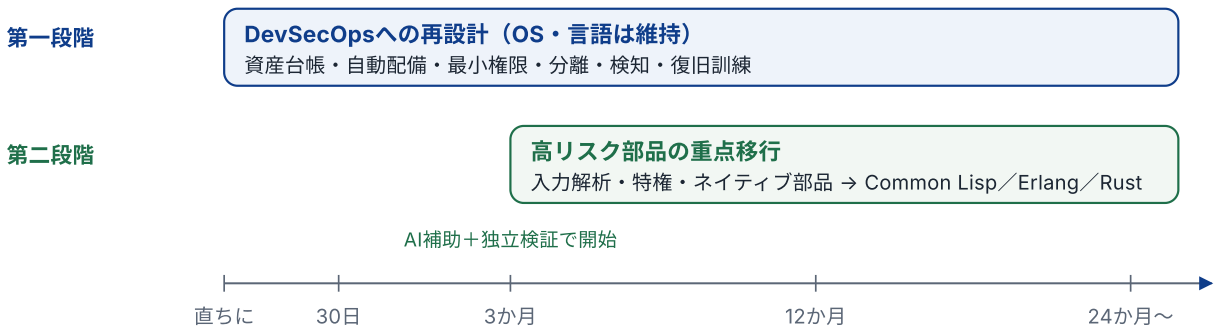

図4　二段階の政策の工程（期間は案）

12.2　公表から72時間の行動計画

PHP2Shellのように、基盤ソフトウェアの重大な脆弱性が公表された場合に、各府省・自治体が従うべき標準の行動計画を平時に定めておく。

時間	行動
予告〜0時間	資産台帳から対象を抽出し、責任者を招集。WAF等の暫定防御を有効化。停止の判断基準（公開範囲、取り扱う情報、修正の可否）を確認
0〜6時間	修正版を取り込んだ成果物を構築・検査し、重要度の高い公開サービスから段階的に配備。修正できないものは機能縮小・アクセス制限・停止を選択
6〜24時間	全対象の稼働中の版と設定を検証。公表前後のログを保全し、侵害の兆候を調査。必要に応じて資格情報を失効
24〜72時間	例外の期限と代替策を記録。国民向けに、脆弱性・試行・侵害・停止を区別して説明。追加の注意喚起（今回の10月23日のような）を反映

医療、交通、防衛などの分野では、一斉の自動更新が別の安全上の事故を生む場合がある。限られた範囲での先行配備、機能停止時の代替手段、既知の安全な構成への復旧を設計する。ただし、脆弱な旧版への無条件の切戻しは避け、隔離や機能の縮小と組み合わせる。

12.3　既存の制度との整合

本稿の提案は、各国で既に進む制度と整合させて導入できる。

国・地域	既存の制度	本稿の提案との関係
米国	NIST SSDF（SP 800-218）、既知の悪用された脆弱性（KEV）への対処期限を定めるBOD 22-01（原則2週間）、メモリ安全性ロードマップの要請[37,39,71,72]	修正配備時間の指標と、言語移行の計画を調達条件に組み込む
欧州連合	サイバーレジリエンス法（規則2024/2847）。2026年9月11日から、製造者は悪用されている脆弱性について24時間以内の早期警告、72時間以内の通知、是正措置が利用可能になってから14日以内の最終報告を求められる[73]	PHP2Shellは報告義務の適用開始後の最初期の大規模事案の一つであり、資産台帳と部品表がなければ期限内の報告は難しい
日本	2025年7月の国家サイバー統括室の設置、いわゆる能動的サイバー防御法（令和7年法律第42号）の主要部分の2026年10月1日施行（基幹インフラ事業者の報告等）[74]。10月24日には、VPS事業者が自社サービスへの攻撃を能動的サイバー防御の「発動第1号」事案と位置付け、政府に攻撃基盤への措置を要請した[82,89]	行政機関・自治体・基幹インフラに共通する資産台帳、修正配備時間、復旧訓練の基準を整備する。攻撃基盤への措置は、被害を受けた組織自身の封じ込めと復旧の能力を代替しない

12.4　国際協調と基盤ソフトウェアへの投資

今回は、修正の公表前に情報が漏えいし、誤った情報や攻撃コードが流通した。[52,55] 各国政府は、重大な脆弱性の調整された公開（協調開示）を支える体制、公開前に重要インフラへ修正を届ける手順、公表直後に参照できる共通の緩和策を、国際的に整備すべきである。また、セキュリティ研究者の整理によれば、今

回の問題の兆候は2024年の外部監査で既に見つかっていたが、修正方針の合意が難航して停滞していた。[52] 世界中の行政が依存する基盤ソフトウェアの保守と監査に、公的な資金を継続的に投じることは、安全保障上の合理的な投資である。

13　AIを使った置換を安全に進める

AIは、Cなどで書かれたコードをCommon Lisp、Erlang、Rustへ翻訳する補助として使える。しかし、コンパイルが通ることは意味の保存を保証しない。整数の幅、符号、文字コード、時刻、並行処理、異常終了、通信規約の互換性、データベースの取引の意味が変わり得る。旧実装との比較試験だけに頼ると、旧実装の欠陥を「正解」として固定してしまう。

まず、入出力の仕様、権限、資源の上限、エラー時の約束を人間が定義し、正常な入力と悪意ある入力の双方で試験する。性質に基づく試験、ファジング、状態機械と重要な不変条件の形式的な検証、独立したレビューを併用する。AIが生成したコードに、生のポインタ操作、`unsafe`、FFI、動的な評価、外部コマンドの実行を、審査なしに追加させない。

高リスクの境界から小さく置き換え、既存のサービスと並行して結果を比較する。生成AIに本番の秘密情報や配備の資格情報を渡さず、変更の提案と本番の承認を分離する。採用する言語は、技術的な特性に加え、保守者の確保、ライブラリの供給、性能目標、検証のしやすさで選ぶ。Common LispとErlangを単に「新しい言語」と呼ぶ必要はない。重要なのは年代ではなく、安全な抽象化と運用可能性である。

今回の事件では、欠陥の発見の一部にAIを用いた解析が使われた。[52,55] 攻撃側については、欧州の政府機関への侵入後の活動にAIエージェントの兆候があったと一時報告されたが、PHP2Shellによる初期侵入とは確認できないとして撤回されている。[53] それでも、欠陥の発見を加速する同じ能力は、攻撃者も利用し得る。攻撃の速度が上がる以上、防御側も、AIを組み込んだ検査、翻訳、試験の生成を、責任と検証の仕組みの中で活用する必要がある。

14　結論：全面刷新を待たず、安全に変えられる能力を制度化する

今回の問題を、「古い言語だから危険」あるいは「コンテナに入れれば安全」という二者択一に還元すべきではない。基盤の欠陥、要求の状態の寿命、実行権限、秘密情報の置き場所、修正の遅れ、そして資産を把握していないことが組み合わさって、社会的な損失が生じた。

政府の第一の仕事は、既存のLinux・UNIX系OSやPHP等を使うサービスであっても、資産を把握し、修正を確実かつ迅速に配備し、侵害の到達範囲を狭め、継続性を検証できるDevSecOpsの体制を作ることである。これはOSや言語を変えずに実行でき、被害の確率と規模を掛け算で減らす。第二の仕事は、AIを補助として、外部入力を扱う高リスク部品から安全な言語・ランタイムへの重点的な移行を進め、同じ種類の欠陥の再生産を減らすことである。

両者を並行させ、予算と責任と検証結果を結び付ける。保証すべきなのは「絶対に侵害されない」という宣言ではなく、危険を継続的に減らす能力と、その成果を説明できる証拠である。

付録A　防御コードの読み解き（実行確認済み）

本付録のコードは教材用であり、実際のPHPの修正パッチや攻撃コードではない。メモリ不足は論理的な予算として模擬しており、実際のOOM処理ではない。すべてのコードは筆者の作成環境（Ubuntu 24.04 LTS）で、次の処理系によりコンパイル・実行し、掲載した出力を得た：gcc 13.3.0（AddressSanitizer）、rustc 1.97.0、GNU CLISP 2.49.95+（`-ansi`指定）、Erlang/OTP 28、PowerShell 7.4.6（Linux版）。YAMLはPyYAMLで構文を、シェルスクリプトは`bash -n`で構文を確認した。実運用では採用する処理系で試験し、時間制限、入出力、並行性、外部への副作用を追加で検証すること。

A.1　C言語とRust：同じ誤りの扱いの違い（コード1・2）

箇所	意味
コード1 (1)	`malloc(16)`で16バイトの領域を借り、そのアドレスを`session`に保持する。
コード1 (2)	`free(session)`で領域を返す。しかし変数`session`には古いアドレスが残ったままで、言語はそれを無効にしない。
コード1 (3)	返却済みの領域を読む。言語規格上の未定義動作であり、コンパイラは通常これを拒否しない。AddressSanitizerを付けた場合に限り、実行時に「heap-use-after-free」として報告される。
コード2 (1)〜(2)	Rustでは`String`の所有者は変数`session`である。`drop(session)`で所有権が関数へ移り、領域が解放される。
コード2 (3)	所有権を失った変数の使用は、コンパイラの借用検査によって拒否される（エラーE0382）。実行可能ファイルは作られない。

政策上の意味：Cでも検査ツールで誤りを見つけられるが、それは試験がその経路を通った場合に限られる。Rustでは、安全なコードの範囲でこの種の誤りが構造的に排除される。調達では、「検査ツールをどの工程で必ず使うか」と「新規開発をどの言語で行うか」の双方を条件にできる。

A.2　Common Lisp：unwind-protect（コード3）

箇所	意味
`*open-buffers*`	確保中の一時バッファの数を数える観測用の変数。後片付けが行われれば0に戻る。

箇所	意味
`parse-multipart-data`	要求が`:malformed`なら、解析の途中で失敗したことを模擬してエラーを通知する。
`unwind-protect`	第一引数（本体）が正常に終わっても、エラー等で脱出しても、第二引数以降（後片付け）を必ず実行する。
`handler-case`	上位でエラーを受け止め、要求の拒否（`:rejected`）に変換する。
出力	`(:PARSED 0)`と`(:REJECTED 0)`。成功時も失敗時も、確保中のバッファは0に戻っている。

A.3　Common Lisp：with-open-file（コード4）

標準マクロ`with-open-file`は、本体を抜けるときにファイルを必ず閉じる。異常な脱出の場合は`:abort t`を付けて閉じる。[78] コメント中の展開は概念を示すもので、実際の展開形は処理系によって異なる。

コード4　Common Lisp：with-open-fileとその概念上の展開　`l2_open_file.lisp`

```
;;; with-open-file：ファイルを必ず閉じる標準マクロ
(defvar *seen* nil)                       ; 使ったストリームを後で確認する

(defun read-first-line (path)
  (with-open-file (stream path :direction :input)
    (setf *seen* stream)
    (read-line stream)))

;; 概念上の展開（細部は処理系ごとに異なる）
;; (let ((stream (open path :direction :input)) (abort t))
;;   (unwind-protect
;;        (multiple-value-prog1 (progn (setf *seen* stream) (read-line stream))
;;          (setq abort nil))                ; 本体が正常に終わった印
;;     (when stream (close stream :abort abort))))  ; 異常時は :abort t で閉じる

(defun demo (path)
  (with-open-file (out path :direction :output :if-exists :supersede)
    (write-line "first line" out))
  (print (read-first-line path))          ; => "first line"
  (print (open-stream-p *seen*))          ; => NIL（閉じている）
  (print (handler-case
             (with-open-file (stream path)
               (setf *seen* stream)
               (error "failure inside the body"))
           (error () :handled)))          ; => :HANDLED
  (print (open-stream-p *seen*)))         ; => NIL（エラー時も閉じている）
```

実行結果（GNU CLISP 2.49.95+）

```
"first line"
NIL
:HANDLED
NIL
```

出力	意味
`"first line"`	ファイルの1行目を読めた。
`NIL`（2行目）	本体を正常に抜けた後、ストリームは閉じている。
`:HANDLED`	本体の途中でエラーが起き、上位で処理された。
`NIL`（4行目）	エラーで脱出した場合も、ストリームは閉じている。

A.4　Common Lisp：条件システムとリスタート（コード5）

Common Lispの条件システムでは、エラーを検出した下位の関数が「回復の選択肢（リスタート）」を定義し、上位のハンドラがスタックを巻き戻さずに状況を調べてから、どの選択肢を使うかを決める。ここでは、チャンク（受信データの断片）3の解析で論理的な予算超過が起き、方針に応じて「キャッシュを解放して再試行」または「要求全体を拒否」を選ぶ。

コード5　Common Lisp：方針は上位が決め、回復は現場で行う　`l3_restart.lisp`

```lisp
;;; 条件システムとリスタート：上位が方針を選び、下位の現場で実行する
;;;  （論理的なメモリ予算のシミュレーション。実際のOOM処理ではない）
(define-condition memory-limit-exceeded (error)
  ((requested-size :initarg :size :reader requested-size))
  (:report (lambda (condition stream)
             (format stream "Memory limit exceeded while requesting ~A bytes."
                     (requested-size condition)))))

(defvar *cache-bytes*)                    ; 解放できるキャッシュ量（論理値）

(defun parse-chunk (chunk-id)
  (let ((retries 0))
    (loop
      (format t "[Parser] Processing chunk ~A...~%" chunk-id)
      (restart-case
          (progn
            (when (and (= chunk-id 3) (> *cache-bytes* 0))
              (error 'memory-limit-exceeded :size 1024))  ; 現場で通知
            (return :ok))                                 ; 成功したら終了
        (free-cache-and-retry ()                ; 選択肢1：状態を変えて再試行
          :report "Free the cache and retry this chunk."
          :test (lambda (c) (declare (ignore c)) (< retries 1)) ; 1回まで
```

```lisp
          (incf retries)
          (format t "[Restart] Freeing ~A bytes of cache.~%" *cache-bytes*)
          (setf *cache-bytes* 0))))))           ; 状態を変えてから戻る

(defun process-request (n-chunks)
  (restart-case
      (progn
        (dotimes (i n-chunks)
          (parse-chunk (1+ i)))
        :accepted)
    (reject-request ()                          ; 選択肢2：要求全体を拒否
      :report "Reject the whole request."
      (format t "[Restart] Request rejected; no partial result is kept.~%")
      :rejected)))

(defun run-with-policy (policy &key (cache 4096))
  (let ((*cache-bytes* cache))
    (handler-bind                               ; 上位：方針を決める
        ((memory-limit-exceeded
           (lambda (c)
             (format t "[Handler] ~A Policy: ~A~%" c policy)
             (let ((retry (find-restart 'free-cache-and-retry c)))
               (if (and (eq policy :retry) retry)
                   (invoke-restart retry)
                   (invoke-restart 'reject-request))))))
      (process-request 5))))
```

実行結果：(run-with-policy :retry) と (run-with-policy :reject)

```
[Parser] Processing chunk 1...
[Parser] Processing chunk 2...
[Parser] Processing chunk 3...
[Handler] Memory limit exceeded while requesting 1024 bytes. Policy: RETRY
[Restart] Freeing 4096 bytes of cache.
[Parser] Processing chunk 3...
[Parser] Processing chunk 4...
[Parser] Processing chunk 5...
=> :ACCEPTED

[Parser] Processing chunk 1...
[Parser] Processing chunk 2...
[Parser] Processing chunk 3...
[Handler] Memory limit exceeded while requesting 1024 bytes. Policy: REJECT
[Restart] Request rejected; no partial result is kept.
=> :REJECTED
```

箇所	意味
`define-condition`	「メモリ上限超過」という種類の条件を定義する。要求された量を保持し、読みやすい説明文を生成する。

箇所	意味
`restart-case`（`parse-chunk`内）	現場で使える回復の選択肢`free-cache-and-retry`を定義する。
`:test`	再試行が1回未満のときだけこの選択肢を有効にする。無限の再試行を防ぐ上限である。
`(setf *cache-bytes* 0)`	再試行の前に状態を変える（キャッシュを解放する）。状態を変えずに再試行しても、同じ失敗を繰り返すだけである。
`reject-request`	要求全体を拒否する選択肢。途中まで解析した部分を「正常な結果」として残さない。
`handler-bind`	上位のハンドラ。巻戻しの前に呼ばれ、方針と、再試行が可能か（`find-restart`）を見て選択肢を呼び出す。
出力（:retry）	チャンク3で失敗→ハンドラ→キャッシュ解放→チャンク3を再処理して成功→4、5と続き`:ACCEPTED`。
出力（:reject）	チャンク3で失敗→ハンドラ→要求を拒否し`:REJECTED`。チャンク4、5は処理しない。

技術ノート：再試行には「状態の変化」と「上限」が必要

筆者の当初のメモでは、リスタートの中で同じ関数を再帰的に呼び直すだけの例を用いていた。その版をGNU CLISPで実行したところ、キャッシュの状態が変わらないため同じ条件が繰り返し通知され、約3,500行の出力の後にスタックあふれで停止した。コード5は、再試行の前に状態を変え、再試行の回数に上限を設け、回復できない場合は要求全体を拒否するよう改めたものである。また、受信データの一部を読み飛ばして正常な取引に見せかける設計は、改ざんや不整合の原因になるため採らない。

A.5　Rust：予算超過を通常のエラーとして返す（コード6）

コード6　Rust：上限検査と失敗し得る確保　r1_bounded.rs

```rust
use std::collections::TryReserveError;

#[derive(Debug)]
enum InputError {
    TooLarge,                       // 上限超過：要求を拒否する
    Capacity(TryReserveError),      // 確保失敗：異常終了せずエラーで返す
}

fn copy_bounded(input: &[u8], limit: usize) -> Result<Vec<u8>, InputError> {
    if input.len() > limit {
        return Err(InputError::TooLarge);       // 確保の前に検査
    }
    let mut owned = Vec::new();
    owned
        .try_reserve_exact(input.len())         // 失敗し得る確保API
        .map_err(InputError::Capacity)?;
    owned.extend_from_slice(input);
    Ok(owned)
}

fn main() {
    for (input, limit) in [(&b"abcd"[..], 3), (&b"abc"[..], 3)] {
        match copy_bounded(input, limit) {
            Ok(v) => println!("accepted: {} bytes", v.len()),
            Err(InputError::TooLarge) => println!("rejected: too large"),
            Err(InputError::Capacity(e)) => println!("rejected: {e}"),
        }
    }
}
```

実行結果（rustc 1.97.0）

```
rejected: too large
accepted: 3 bytes
```

箇所	意味
`enum InputError`	失敗の種類を型として列挙する。呼び出し側はすべての場合を処理するようコンパイラに求められる。
`input.len() > limit`	領域を確保する前に、入力の大きさを上限と比べる。
`try_reserve_exact`	確保に失敗したとき、プロセスを異常終了させずにエラーとして返すAPI。
`?`	エラーなら、その場で呼び出し元へ返す。

この例では、コピー前の入力自体は既にメモリ上に存在するため、通信の入口でのストリーミングによる上限の代わりにはならない。また`try_reserve`は、後段でのOSによる強制終了まで防ぐものではない。[28,29]

A.6　Rust：後片付け（Drop）が走る場合と走らない場合（コード7）

コード7　Rust：Dropによる後片付けとpanicの設定　r2_drop.rs

```rust
struct Parser {
    name: &'static str,
}

impl Drop for Parser {                  // 所有者が消えるときに必ず呼ばれる後始末
    fn drop(&mut self) {
        println!("cleanup: {}", self.name);
    }
}

fn parse(fail: bool) -> Result<(), String> {
    let _p = Parser { name: "multipart" };
    if fail {
        return Err("budget exceeded".to_string()); // 早期returnでもDropは実行
    }
    Ok(())
}

fn main() {
    println!("{:?}", parse(true));

    let r = std::panic::catch_unwind(|| {         // panic=unwind の場合
        let _p = Parser { name: "during-panic" };
        panic!("simulated fatal error");           // 巻戻し中にDropが走る
    });
    println!("panic caught: {}", r.is_err());
}
```

実行結果（既定のpanic=unwind）

```
cleanup: multipart
Err("budget exceeded")
cleanup: during-panic
panic caught: true
```

実行結果（-C panic=abort でコンパイル）

```
cleanup: multipart
Err("budget exceeded")
exit status: 134
```

出力	意味
`cleanup: multipart`	早期の `return` で関数を抜けても、`Drop` による後片付けが実行された。
`cleanup: during-panic`	既定の設定では、panic時にスタックが巻き戻され、途中の値の `Drop` が実行された。
`exit status: 134`	`panic=abort` では巻戻しが行われず、`cleanup: during-panic` は表示されないままプロセスが中断した（134はSIGABRTによる終了）。

政策上の意味：「Rustなら必ず後片付けされる」とは言えない。ビルドの設定やOSによる強制終了で保証は変わる。調達仕様では、言語名だけでなく、ビルド設定と異常終了時の扱いまで確認する必要がある。

A.7　Erlang：子プロセスの失敗を親が観測する（コード8）

コード8　Erlang：要求ごとの軽量プロセスと監視　`e1_monitor.erl`

```erlang
-module(request_demo).
-export([run/2]).

%% 要求ごとに軽量プロセスを作り、失敗は親が「DOWN」通知で観測する
run(Data, Limit) when is_binary(Data),
                      is_integer(Limit), Limit >= 0 ->
    Parent = self(),
    {Pid, Ref} = spawn_monitor(fun() ->
        case byte_size(Data) =< Limit of
            true  -> Parent ! {self(), accepted};
            false -> exit(request_too_large)      % 子プロセスだけが終了
        end
    end),
    receive
        {Pid, accepted} ->
            erlang:demonitor(Ref, [flush]),
            {ok, accepted};
        {'DOWN', Ref, process, Pid, Reason} ->    % 親は理由付きで受け取る
            {error, Reason}
    end.
```

実行結果：run(<<1,2,3>>, 2) と run(<<1,2>>, 2)（Erlang/OTP 28）

```
{error,request_too_large}
{ok,accepted}
```

箇所	意味
`spawn_monitor`	要求を処理する軽量プロセスを作り、同時にその終了を監視する。
`exit(request_too_large)`	上限を超えた要求では、子プロセスだけが理由付きで終了する。親や他の要求には影響しない。
`'DOWN'` メッセージ	子の終了は、理由を含むメッセージとして親に届く。親は再起動や拒否を選べる。
`demonitor(Ref, [flush])`	正常終了時に監視を解除し、遅れて届く通知も破棄する。

これはOTPのsupervisorの実装ではなく、実際のメモリ枯渇の試験でもない。金銭の処理などを自動で再試行するには、二重処理を防ぐための識別子（冪等性キー）や取引の記録が別途必要になる。[26,27]

付録B　既存のPHPを守る配備・運用テンプレート

以下は設定の意図を示すテンプレートである。SHAやダイジェストなどのプレースホルダーは審査済みの値に置き換える。構文の確認は行ったが、GitHub、Kubernetes、Falcoでの統合的な稼働試験は行っておらず、完成した本番の配備一式ではない。B.7とB.8のスクリプトはテスト環境で実行して出力を確認した（条件は各節に記す）。

B.1　PHPの設定（コード9）

修正版の適用と10月23日の注意喚起を前提に、不要なアップロード進捗機能を無効化する。数値は説明のための例であり、業務ごとに負荷試験で決める。アプリケーション内の`ini_set`では、要求の開始前に行われる解析を保護できない。[1,22]

コード9　php.ini：修正版に追加する設定の例　b1_php.ini

```
; 修正版へ更新した環境の追加設定例（これ単独では修正にならない）
; 実際に動いているSAPI（PHP-FPM等）の設定ファイルに書くこと
session.upload_progress.enabled = Off  ; 10月23日の注意喚起に対応

post_max_size = 8M            ; POST本文の上限（例示値）
upload_max_filesize = 2M      ; アップロード1件の上限（例示値）
max_input_vars = 1000         ; 受け付ける入力変数の数
max_input_nesting_level = 32  ; 入力配列の入れ子の深さ
max_file_uploads = 4          ; 1要求あたりのファイル数

; memory_limit について万人向けの推奨値は示さない
; 起動時設定を変えたら全ワーカーを再起動する
```

項目	意味と注意
`session.upload_progress.enabled`	アップロード進捗の追跡機能。既定で有効。CVE-2026-115560の修正までは無効化が案内されている。
`post_max_size`、`upload_max_filesize`	受け付ける本文とファイルの大きさの上限。業務上必要な最小値にする。
`max_input_vars`、`max_input_nesting_level`	入力変数の数と入れ子の深さの上限。解析の負荷を抑える。
全ワーカーの再起動	起動時の設定はワーカーを入れ替えないと反映されない。

B.2　Webサーバーでの上限（コード10）

コード10　nginx：PHPへ渡す前の本文サイズの制限　`b2_nginx.conf`

```
# PHPへ渡す前に本文サイズを制限する（多層防御の一部）
client_max_body_size    8m;     # 超過した要求は413で拒否
client_body_buffer_size 128k;   # 超える本文は一時ファイルへ書かれる
# 注意：これらはPHP内部のメモリ使用量の上限ではない
```

`client_max_body_size`を超える要求は413エラーで拒否され、`client_body_buffer_size`を超える本文は一時ファイルに書かれる。[47] これらは多層防御の一部であり、PHP内部のメモリ使用量の上限ではない。

B.3　資産の棚卸し（コード11）

コード11　シェル：PHP資産の棚卸しの最小例 b3_inventory.sh

```
#!/usr/bin/env bash
# PHP資産の棚卸し（最小例）。結果は資産台帳へ記録する
set -euo pipefail

echo "== 導入済みパッケージ =="
if command -v dpkg-query >/dev/null; then
  dpkg-query -W -f='${db:Status-Abbrev} ${Package} ${Version}\n' 'php*' \
    2>/dev/null | awk '$1 == "ii" {print $2, $3}'   # 導入済みのみ
elif command -v rpm >/dev/null; then
  rpm -qa 'php*' || true
fi

echo "== CLIの版と設定（FPM/Apacheとは異なる場合がある）=="
if command -v php >/dev/null; then
  php -r 'echo PHP_VERSION, " ", PHP_SAPI, PHP_EOL;'
  php -r 'echo "upload_progress=", ini_get("session.upload_progress.enabled"), PHP_EOL;'
fi

echo "== 稼働中のPHPプロセス =="
ps -eo pid,user,args | grep -E '[p]hp-fpm|[h]ttpd|[a]pache2' || true
```

作成環境でこのスクリプトを実行すると、`php8.3-cli 8.3.6-0ubuntu0.24.04.13`などが表示された。版番号8.3.6は修正版8.3.36より古く見えるが、パッケージの変更履歴（`/usr/share/doc/php8.3-common/changelog.Debian.gz`）には、CVE-2026-109618と109619の修正が2026年10月19日付で記録されていた。[51] 判定には版番号ではなく修正履歴を用いる。また、CLIの設定はPHP-FPMやApacheの設定と異なる場合があるため、稼働中のSAPIごとに確認する。

B.4　CIの検査関門（コード12）

サードパーティのActionを可変のタグ（masterなど）で参照せず、審査済みのコミットに固定する。検査ツールも版、チェックサム、配布経路を確認する。修正のない脆弱性を一括して除外する`ignore-unfixed`は使わない。必要な例外は、根拠、代替策、責任者、期限を記録する。[40,42]

コード12　GitHub Actions：検査に失敗したら先へ進ませない　b4_actions.yml

```yaml
name: Security gate
on:
  pull_request:
  push:
    branches: [main]
permissions:
  contents: read                    # 既定の権限を読み取りのみに絞る
jobs:
  security-gate:
    runs-on: ubuntu-latest
    timeout-minutes: 20
    steps:
      - uses: actions/checkout@<REVIEWED_FULL_COMMIT_SHA>  # 可変タグでなく固定
        with:
          persist-credentials: false
      # ここに版を固定し検証済みのTrivy導入手順を入れる
      - name: Scan source and dependencies
        shell: bash
        run: |
          set -euo pipefail
          command -v trivy
          trivy fs --scanners vuln,secret,misconfig \
            --severity HIGH,CRITICAL --exit-code 1 .
      - name: Build and scan candidate
        shell: bash
        run: |
          set -euo pipefail
          docker build --pull -t service:candidate .
          trivy image --scanners vuln \
            --severity HIGH,CRITICAL --exit-code 1 service:candidate
      # 承認済みの別ジョブが、検査したダイジェストを本番へ昇格する
      # PRのジョブには本番の資格情報を渡さない
```

箇所	意味と注意
`permissions: contents: read`	ジョブに与える権限を読み取りのみに絞る。
`@<REVIEWED_FULL_COMMIT_SHA>`	外部のActionを審査済みのコミットに固定し、すり替えを防ぐ。
`--exit-code 1`	重大・高の脆弱性があればジョブを失敗させる。
必須の検査	終了コード1だけではマージは止まらない。ブランチ保護またはルールセットでこのジョブを必須にし、無審査の迂回権限を抑える。[41]
PRのジョブ	PRのDockerfile自体が攻撃に使われ得るため、秘密情報のない使い捨ての実行環境で動かす。

箇所	意味と注意
昇格	本番には、別に再構築したものではなく、検査したダイジェストと同一の成果物を承認済みのジョブで昇格させる。

既知の脆弱性データベースに未収録の欠陥や誤認識は残るため、検査の成功は完全な安全を意味しない。

B.5　実行権限とネットワークの限定（コード13・14）

UID、書き込み先、容量、イメージはアプリケーションに合わせて検証する。非rootでの実行に対応していない既存のイメージにそのまま適用すると、起動に失敗する場合がある。CPUとメモリの制限は容量計画と併用する。[35]

コード13　Kubernetes：Podテンプレートの一部　b5_pod.yaml

```yaml
# Podテンプレートの一部（完全なDeploymentではない）
spec:
  automountServiceAccountToken: false   # 不要なAPI資格情報を持たせない
  securityContext:
    runAsNonRoot: true                  # rootで動かさない
    runAsUser: 10001
    runAsGroup: 10001
    fsGroup: 10001
  containers:
    - name: php-app
      image: registry.example.gov/service@sha256:<APPROVED_DIGEST>  # 承認済みの実体
      securityContext:
        allowPrivilegeEscalation: false # 権限昇格を禁止
        readOnlyRootFilesystem: true    # 改ざん・バックドア設置を難しくする
        capabilities:
          drop: [ALL]                   # 特権機能をすべて外す
        seccompProfile:
          type: RuntimeDefault          # 使えるシステムコールを絞る
      resources:
        requests:
          memory: 256Mi
          cpu: 250m
        limits:
          memory: 512Mi                 # コンテナ全体のメモリ上限
          cpu: '1'
      volumeMounts:
        - name: scratch
          mountPath: /tmp
  volumes:
    - name: scratch
      emptyDir:
        sizeLimit: 64Mi
```

コード14　Kubernetes：既定で通信を拒否するポリシー　b6_netpol.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: public-services
spec:
  podSelector: {}                 # 名前空間内のすべてのPodが対象
  policyTypes: [Ingress, Egress]  # 受信も送信も既定で拒否
# 入口・DNS・必要なAPIだけを、別途審査した許可ルールで開ける
# NetworkPolicyを実際に強制するネットワークプラグインが必要
```

NetworkPolicyを実際に強制するネットワークプラグイン（CNI）が必要であり、宣言だけでは強制されない。DNS、入口、必要なデータベースやAPIだけを別のルールで許可する。まず試験環境で検証し、許可ルールなしの一括投入で行政サービスを止めないようにする。[36]

B.6　振る舞いの検知と安全な対応（コード15）

コード15　Falco：Webワーカーが直接シェルを起動したら警告　b7_falco.yaml

```yaml
# Falco標準の spawned_process / container マクロを前提とする
- rule: Shell directly spawned by web worker
  desc: Investigate unexpected shell execution by a web worker
  condition: >
    spawned_process and container
    and proc.pname in (php-fpm, php-fpm8.2, php-fpm8.3,
                       php-fpm8.4, php-fpm8.5, apache2, httpd)
    and proc.name in (sh, bash, dash, zsh, ksh)
  output: >
    Unexpected shell parent=%proc.pname process=%proc.name
    container=%container.id user=%user.name
  priority: WARNING
  tags: [execution, web]
# 実際のプロセス名と正当な子プロセスを自環境で試験すること
# このルールは警告を出すだけで、自動で遮断・削除はしない
```

このルールは直接の親プロセス名に依存するため、シェルを起動しない攻撃や、別の親を経由する攻撃を見逃す。正常な子プロセスの起動も検知し得る。Falco自体がUAFの発生前に攻撃を遮断するわけではない。[43] SIEM等への連携には通知先の設定が要り、連携ツールを導入しただけで安全な自動削除が完成するわけでもない。

対応の手順は、記録と相関 → 疑わしい対象の通信の隔離 → 証拠の保全 → 資格情報の失効 → 修正済みの成果物での再配備 → データの整合性の確認、とする。緊急の遮断を優先すべき場合もあるが、全件の自動削除による証拠の喪失やサービスの妨害を避ける。障害を意図的に注入する試験（カオス試験）は、まず隔離した環境で行い、本番では許可した範囲、停止条件、代替サービスを用意して実施する。

B.7　更新が稼働中のプロセスに反映されたか（Linux、コード16）

第6.3節で述べたとおり、パッケージを更新しても、既に動いているプロセスは古い実行ファイルやライブラリを使い続ける。次のスクリプトは、PHP関連のプロセスごとに、実行ファイルと読み込み中のPHPライブラリが削除済み（更新で置き換え済み）でないかを調べ、再起動が必要なものをSTALEと表示する。

コード16　シェル：古い実体のまま動くPHP関連プロセスを検出する　　b9_running_php.sh

```
#!/usr/bin/env bash
# 稼働中のPHP関連プロセスが、更新で置き換わった古い実行ファイルや
# PHPライブラリを使い続けていないかを確認する（読み取りのみ。root推奨）
set -uo pipefail
pattern='^(php[0-9.]*|php-fpm[0-9.]*|php-cgi[0-9.]*|lsphp[0-
9]*|frankenphp|apache2|httpd)$'

for dir in /proc/[0-9]*; do
  pid=${dir#/proc/}
  comm=$(cat "$dir/comm" 2>/dev/null) || continue
  [[ $comm =~ $pattern ]] || continue                    # PHP関連のプロセスだけ
  [[ $pid == "$$" ]] && continue
  exe=$(readlink "$dir/exe" 2>/dev/null || echo '?')
  # 削除済み（＝更新で置き換えられた）PHPライブラリを読み込んだままか
  stale_lib=$(grep -E '/(libphp|php)[^ ]*\.so[^ ]* \(deleted\)$' \
                "$dir/maps" 2>/dev/null | head -n 1)
  status=OK
  [[ $exe == *' (deleted)' || -n $stale_lib ]] && status=STALE  # 再起動が必要
  user=$(stat -c %U "$dir")
  printf '%-6s %-7s %-10s %-10s %s\n' "$status" "$pid" "$user" "$comm" "$exe"
done
```

実行結果（テスト環境で更新を模擬。作業ディレクトリのパスは「…」で省略）

```
STALE  3425    root       php8.3     …/php8.3 (deleted)
OK     3426    root       php        /usr/bin/php8.3
STALE  3427    root       apache2    …/apache2
```

箇所	意味
`/proc/[0-9]*`	稼働中の全プロセスを調べる。他のユーザーのプロセスを読むにはroot権限が要る。
`pattern`	PHPのCLI、PHP-FPM、php-cgi、LiteSpeedのlsphp、FrankenPHP、Apacheを対象とする。mod_phpはApacheのプロセスに読み込まれるため、Apacheも含める。
`readlink "$dir/exe"`	実行ファイルの実体。更新で置き換えられていれば、末尾に「(deleted)」が付く。

箇所	意味
`maps` の `(deleted)`	読み込み中のPHPライブラリ（mod_phpのlibphpなど）が置き換え済みかを調べる。
`STALE` ／ `OK`	STALEは再起動（ワーカーの入替え）が必要。OKは現在のファイルで動いている。

テストでは、PHPの実行ファイルをコピーして起動した後にファイルを置き換え、パッケージの更新を模擬した（1行目）。また、apache2という名前で起動したテスト用のプロセスにlibphp8.3.soという名前のライブラリを読み込ませた後に置き換え、mod_phpの状態を模擬した（3行目）。こちらは実行ファイル自体を置き換えていないためパスに「(deleted)」は付かないが、ライブラリの検査でSTALEと判定された。通常の `/usr/bin/php` で動くプロセスはOKである（2行目）。実際のApacheとPHP-FPMでの稼働試験は行っていない。STALEのプロセスを持つサービスは、証拠保全の要否を判断したうえで再起動する。

B.8　Windows Server：IISが呼び出すPHPを特定する（コード17）

次のPowerShellスクリプトは、IISの設定ファイル（applicationHost.config）を読み、*.phpのハンドラーが渡す実行ファイルと、FastCGIアプリケーションとして登録されたPHPの実体、php.iniの場所（環境変数PHPRC）、表示される版、SHA-256ハッシュを出力する。読み取りのみで、設定は変更しない。

コード17　PowerShell：IISの設定から実際のPHPと版・ハッシュを記録する　b8_iis_php.ps1

```
# IISが実際に呼び出すPHPを applicationHost.config から特定する（読み取りのみ）
param(
  [string]$Config = "$env:windir\System32\inetsrv\config\applicationHost.config"
)
[xml]$cfg = Get-Content -LiteralPath $Config -Raw

# (1) *.php をどの実行ファイルに渡すか（ハンドラーマッピング）
foreach ($h in $cfg.SelectNodes("//handlers/add[@path='*.php']")) {
  'handler {0}: {1}' -f $h.name, $h.scriptProcessor
}

# (2) FastCGIアプリケーションの実体。表示される版が同じでも
#     配布物の中身は違い得るため、SHA-256も記録する
foreach ($app in $cfg.SelectNodes('//system.webServer/fastCgi/application')) {
  $exe = $app.fullPath
  $rc  = $app.SelectSingleNode("environmentVariables/environmentVariable[@name='PHPRC']")
  'FullPath : {0}' -f $exe
  'PHPRC    : {0}' -f $(if ($rc) { $rc.value } else { '(not set)' })  # php.ini の場所
  'Version  : {0}' -f (& $exe -v | Select-Object -First 1)
  'SHA256   : {0}' -f (Get-FileHash -Algorithm SHA256 -LiteralPath $exe).Hash
}
```

実行結果（PowerShell 7.4.6でテスト用の設定ファイルを読み込み。パスの前半は「…」で省略）

```
handler PHP_via_FastCGI: …/PHP/php-cgi.exe
FullPath : …/PHP/php-cgi.exe
PHPRC    : C:\PHP
Version  : PHP 8.3.6 (cli) (built: Oct 19 2026 15:16:58) (NTS)
SHA256   : 7F1AC2FE89D267364311651C0E7E6651D32EA006C9A701C0CAB68EB2394A67D7
```

テストはLinux上のPowerShell 7.4.6で行い、Windowsの設定形式に倣った設定ファイルを読み込み、php-cgi.exeという名前で配置したPHPのCLIを対象とした。そのため、版の表示は「(cli)」となっている。Windows Serverの実機とIISでの稼働試験は行っていない。サイトのweb.configでハンドラーが上書きされる場合もあるため、IISマネージャーでも確認する。SHA-256は、配布元のダウンロードページに示される値と照合する。10月26日の再ビルドのように、版の表示が同じでも中身が異なる場合があるためである。[85]

付録C　政策担当者のための問答集

問1　古い言語やOSが危険なら、すべて書き換えればよいのではないか。

> 世界中の行政システムが依存するOSと言語処理系を一度に書き換えることは、費用、人材、期間、互換性のいずれの面でも現実的でない。書き換えの途中で新たな欠陥を生む危険もある。本稿は、まずDevSecOpsで被害の確率と規模を下げ（第一段階）、外部入力を扱う高リスク部品から重点的に置き換える（第二段階）ことを提案する。

問2　外国からの攻撃を遮断すれば十分ではないか。

> 十分ではない。攻撃は国内で乗っ取られた機器、窃取された正規アカウント、改ざんされた部品を経由して行われる。通信元の国による判断は補助情報にすぎず、各サービスで認証・認可を行い、侵入されても被害が広がらない構造が必要である（ゼロトラスト）。

問3　WAF（Webアプリケーションファイアウォール）を入れれば安全か。

> 今回、WAFは公表前から有効な防御層となった。しかしWAFを経由しない通信には効かず、攻撃の変形も観測された。WAFは修正の代わりではなく、修正までの時間を稼ぐ層の一つである。

問4　クラウドやVPS、コンテナに移せば解決するか。

> コンテナやサーバーレスは被害の区画化と入替えを容易にするが、実行環境は再利用されることがあり、最初の要求で盗まれた情報は戻らない。権限設定を誤れば区画化も機能しない。仕組みの導入ではなく、権限・検知・復旧を含めて検証された運用が重要である。VPSも同じであり、仮想化の境界は契約者どうしを分けても、各サーバーの内側の欠陥と秘密情報は守らない（第6節）。

問5　サービスを止めたことは誤りだったのか。

> 修正の確実な配備や侵害の有無の確認ができない状況では、予防的な停止は合理的な判断であり得る。問題は、停止以外の選択肢がなかったことにある。資産台帳と迅速な配備の能力があれば、停止の範囲と時間を大きく縮められる。

問6　AIでコードを書き換えれば安く早く安全になるか。

> AIは強力な補助であるが、コンパイルが通ることは意味が保たれていることを保証しない。人間の責任者、仕様の明確化、悪意ある入力を含む試験、形式的な検証、独立したレビューを必須とし、高リスクの部品から小さく置き換える。

問7　効果をどう説明すればよいか。

> 第11節の指標（公開資産の把握率、修正配備時間、越権アクセスの遮断率、復旧試験の成功率など）を導入前後で測定し、公表する。「絶対に安全」ではなく、「危険をこれだけ減らし、これだけ早く回復できる」ことを証拠で示す。

付録D　用語集

用語	意味
CVE／CVSS	脆弱性に付与される共通の識別番号／深刻度を数値で表す評価の方式。評価者によって値が異なることがある。
KEV	米国CISAが公表する、実際に悪用が確認された脆弱性の一覧。連邦機関には対処期限が課される。
ゼロデイ	修正が公表される前に悪用される脆弱性、またはその攻撃。
エンバーゴ	修正の準備が整うまで脆弱性情報を関係者の間に留める取り決め。今回は公表前に破られた。
PoC	攻撃が可能であることを示す実証コード。
OOM／UAF／RCE	メモリ不足／解放済みのメモリの不正な使用／遠隔からの任意のコード実行。三者は同義ではなく、OOMが常にUAFやRCEになるわけではない。
メモリ安全性	確保した範囲の外や解放済みの領域へのアクセスが、言語や実行環境によって防がれている性質。
SAPI／mod_php／PHP-FPM	PHPとWebサーバー等をつなぐ接続方式／Apacheに組み込む方式／独立したプロセス群として動かす方式。
ワーカー	要求を処理する作業用プロセス。性能のため多数の要求に使い回される。

用語	意味
GC／FFI	不要になったメモリを自動で回収する仕組み／他の言語の関数を呼ぶ仕組み（外部ライブラリの危険も取り込む）。
WAF	Webへの通信を検査し、攻撃とみられるものを遮断する装置・サービス。
DevSecOps	開発・セキュリティ・運用を、同じ変更管理と証拠の連鎖で結ぶ継続的な業務方式。
CI／CD	継続的インテグレーション／継続的デリバリー（またはデプロイメント）。
SBOM	ソフトウェア部品表。含まれる部品を把握する資料であり、安全の証明書ではない。
SAST／DAST	ソースコード等を解析する静的検査／動作中のアプリケーションを調べる動的検査。どちらも全欠陥の発見は保証しない。
ダイジェスト	成果物の内容から計算される一意の値。同じ値なら同じ中身である。
ゼロトラスト	ネットワーク上の位置を信頼の根拠とせず、要求ごとに認証・認可する考え方。
最小権限	各部品に、業務に必要な最小限の権限だけを与える原則。
SIEM／SOAR	複数の記録を集約・分析する監視基盤／対応手順を連携・自動化する仕組み。自動化の権限と停止条件も管理対象となる。
RTO／RPO	復旧までの目標時間／許容するデータ損失の時間幅。復旧訓練で実現可能性を測る。
バックポート	新しい版の修正を、版番号を変えずに古い版へ取り込むこと。
VPS	仮想専用サーバー。物理サーバーを仮想化で区切り、利用者ごとに専用のサーバーのように貸し出す形態。ゲストOS内部の更新の責任は、通常は利用者にある。

参考文献

資料の確認日：2026年10月26日（日本時間）。動的な文書は今後更新され得る。URLが変更・制限されている場合は、資料名と識別番号で発行元を確認されたい。番号1〜49は初稿からの文献、50以降は本改訂で追加した文献である。

[1] PHP Group (2026). Security advisory, 21 October 2026; Session upload progress notice, 23 October 2026. PHP News Archive.
https://www.php.net/archive/2026.php

[2] PHP / php-src. RCE via OOM corruption, GHSA-w8hw-w9fx-j79w (CVE-2026-109618); Susceptibility and Mitigation.
https://github.com/php/php-src/security/advisories/GHSA-w8hw-w9fx-j79w

[3] PHP / php-src. RCE via stale global state, GHSA-6wmf-4rv5-p3rp (CVE-2026-109619); Producers, Consumers.
https://github.com/php/php-src/security/advisories/GHSA-6wmf-4rv5-p3rp

[4] JPCERT/CC (2026). PHPの脆弱性に関する注意喚起, at260033.
https://www.jpcert.or.jp/at/2026/at260033.html

[5] Amazon Threat Intelligence (2026). PHP2Shell exploitation identified prior to public disclosure in MadPot.
https://aws.amazon.com/blogs/security/amazon-threat-intelligence-identifies-php2shell-exploitation-prior-to-public-disclosure-in-madpot/

[6] 財務省 (2026年10月21日). PHPの脆弱性対応に伴う一時停止のお知らせ.
https://www.mof.go.jp/public_relations/ohter/20261021231537.html

[7] 財務省 (2026年10月22日). 公開再開のお知らせ.
https://www.mof.go.jp/public_relations/ohter/20261022184327.html

[8] CERT-EU (2026). Critical Vulnerabilities in PHP, Security Advisory 2026-018.
https://cert.europa.eu/publications/security-advisories/2026-018/

[9] GNU C Library (17 April 2024). GLIBC-SA-2024-0004 / CVE-2024-2961: iconv ISO-2022-CN-EXT buffer overflow.
https://www.openwall.com/lists/oss-security/2024/04/17/9

[10] J. Postel (1981). RFC 801: NCP/TCP Transition Plan.
https://www.rfc-editor.org/info/rfc801

[11] R. M. Metcalfe (1973). RFC 602: The Stockings Were Hung by the Chimney with Care.
https://www.rfc-editor.org/rfc/rfc602.html

[12] D. M. Ritchie. The Evolution of the Unix Time-sharing System.
https://www.nokia.com/bell-labs/about/dennis-m-ritchie/hist.html

[13] The Open Group. History and Timeline of UNIX (1969, 1989, 1993 entries).
https://unix.org/unix_history.html

[14] FreeBSD Project. BSD family tree.
https://cgit.freebsd.org/src/tree/share/misc/bsd-family-tree

[15] FreeBSD Project. Explaining BSD.
https://docs.freebsd.org/en/articles/explaining-bsd/index.html

[16] OpenSSH Project. Project History.
https://www.openssh.org/history.html

[17] IBM Japan. Corporate history (2000: investment of $1 billion in Linux).
https://www.ibm.com/jp-ja/about/ibm-japan-history

[18] Carnegie Mellon University. Overview of the Mach Project.
https://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/overview.html

[19] D. Golub, R. Dean, A. Forin, R. Rashid (1990). Unix as an Application Program. USENIX Summer Conference.
https://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/mach3_intro.html

[20] Apple. XNU source repository, README.
https://github.com/apple-oss-distributions/xnu

[21] PHP Group. History of PHP.
https://www.php.net/manual/en/history.php.php

[22] PHP Group. Description of core php.ini directives.
https://www.php.net/manual/en/ini.core.php

[23] LispWorks. Common Lisp HyperSpec: UNWIND-PROTECT.
https://www.lispworks.com/documentation/HyperSpec/Body/s_unwind.htm

[24] LispWorks. Common Lisp HyperSpec: HANDLER-BIND.
https://www.lispworks.com/documentation/HyperSpec/Body/m_handle.htm

[25] LispWorks. Common Lisp HyperSpec: RESTART-CASE.
https://www.lispworks.com/documentation/HyperSpec/Body/m_rst_ca.htm

[26] Erlang/OTP. Robustness.
https://www.erlang.org/doc/system/robustness.html

[27] Erlang/OTP. erl_nif: “A native function that crashes will crash the whole VM.”
https://www.erlang.org/doc/apps/erts/erl_nif.html

[28] Rust Project. The Rustonomicon: What Unsafe Rust Can Do.
https://doc.rust-lang.org/nomicon/what-unsafe-does.html

[29] Rust Project. The Rust Programming Language: Unrecoverable Errors with panic!.
https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html

[30] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, J. White (2000). Formal Analysis of the Remote Agent Before and After Flight. NASA.
https://ntrs.nasa.gov/citations/20000055731

[31] NASA/JPL (7 June 1999). Remote Agent Experiment Meets All Objectives.
https://www.jpl.nasa.gov/news/remote-agent-experiment-meets-all-objectives

[32] AWS. Understanding the Lambda execution environment lifecycle.
https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html

[33] Firecracker Project. README.
https://github.com/firecracker-microvm/firecracker

[34] Kubernetes Project. Security.
https://kubernetes.io/docs/concepts/security/

[35] Kubernetes Project. Pod Security Standards.
https://kubernetes.io/docs/concepts/security/pod-security-standards/

[36] Kubernetes Project. Network Policies.
https://kubernetes.io/docs/concepts/services-networking/network-policies/

[37] NIST (2022). SP 800-218: Secure Software Development Framework (SSDF), Version 1.1.
https://csrc.nist.gov/pubs/sp/800/218/final

[38] NIST (2020). SP 800-207: Zero Trust Architecture.
https://csrc.nist.gov/pubs/sp/800/207/final

[39] CISA, NSA, FBI, ASD's ACSC, CCCS, NCSC-UK, NCSC-NZ, CERT NZ (December 2023). The Case for Memory Safe Roadmaps.

https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps

[40] GitHub. Secure use reference for GitHub Actions.
https://docs.github.com/en/actions/reference/security/secure-use

[41] GitHub. About protected branches.
https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-protected-branches/about-protected-branches

[42] Trivy. Filtering.
https://trivy.dev/docs/latest/configuration/filtering/

[43] Falco. Rules.
https://falco.org/docs/concepts/rules/

[44] PeachPie. Documentation.
https://docs.peachpie.io/

[45] ISO/IEC JTC1/SC22/WG14 (2011). N1570 (C11 committee draft), 7.13.2.1 The longjmp function.
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[46] CMUCL. User's Manual: Credits.
https://cmucl.org/docs/cmu-user/html/Credits.html

[47] NGINX. Module ngx_http_core_module: client_max_body_size, client_body_buffer_size.
https://nginx.org/en/docs/http/ngx_http_core_module.html

[48] Cognitive Research Labs, Inc. 政府プロジェクト：コグニティブリサーチラボラトリィズ株式会社時代の政府プロジェクト.
https://crl.co.jp/government-projects

[49] Cognitive Research Labs, Inc. Government Projects.
https://crl.co.jp/en/government-projects

[50] A. Le Blanc, I. Tovilo, J. Zelenka, N. Dossche, N. Grekas (18 October 2026). Avoid state corruption via bailout. php-src commit 31c0ea7 (PHP-8.5 branch).
https://github.com/php/php-src/commit/31c0ea7d7ff1d1fccb6492e52dab544011628065

[51] Ubuntu (19 October 2026). php8.3 8.3.6-0ubuntu0.24.04.13 (noble-security) changelog: CVE-2026-109618_109619.patch.
https://launchpad.net/ubuntu/+source/php8.3

[52] piyokango (2026). PHPの2件の深刻な脆弱性（CVE-2026-109618、CVE-2026-109619）についてまとめてみた. piyolog (updated through 24 October 2026).
https://piyolog.hatenadiary.jp/entry/2026/10/21/174516

[53] Palo Alto Networks Unit 42 (2026). Threat Brief: Critical Vulnerabilities in PHP: CVE-2026-109618, CVE-2026-109619 (updated 25 October 2026; the 23 October addition on a European government entity was withdrawn).
https://unit42.paloaltonetworks.com/cve-2026-109618-cve-2026-109619/

[54] Canadian Centre for Cyber Security (21 October 2026, updated 23 October). AL26-026: Critical vulnerabilities impacting PHP.
https://cyber.gc.ca/en/alerts-advisories/al26-026-critical-vulnerabilities-impacting-php-cve-2026-109618-cve-2026-109619

[55] Recorded Future Insikt Group (21 October 2026). PHP2Shell Exploitation: Two Critical PHP Flaws Explained.
https://www.recordedfuture.com/blog/php2shell-vulnerability-exploitation

[56] Jiji Press / Nippon.com (21 October 2026). Japan PM Warns of Vulnerabilities in PHP Programming Language.
https://www.nippon.com/en/news/yjj2026102101249/

[57] AAP (22 October 2026). Government websites go dark in software flaw scramble.
https://www.portnews.com.au/story/9373041/government-websites-go-dark-in-software-flaw-scramble/

[58] The Canadian Press / CTV News (21 October 2026). Some government websites taken down in response to security risk: AI minister.
https://www.ctvnews.ca/sci-tech/article/some-government-websites-taken-down-in-response-to-security-risk-ai-minister/

[59] The Hacker News (October 2026). PHP2Shell Exploitation Attempts Continue Amid Suspected N. Korea Global Scanning Activity.
https://thehackernews.com/2026/10/php2shell-exploitation-hits-government.html

[60] PPC Land (2026). Cloudflare blocks php2shell exploits 9 hours before PHP flaws went public.
https://ppc.land/cloudflare-blocks-php2shell-exploits-9-hours-before-php-flaws-went-public/

[61] Kredo (24 October 2026). PHP脆弱性ニュース【2026年10月】国内事例・政府発表・時系列.
https://kredo.jp/media/php-vulnerability-news-2026-october/

[62] Bitsight (21 October 2026). CVE-2026-109618 and CVE-2026-109619 (PHP2Shell): Two Critical PHP Remote Code Execution Flaws.
https://www.bitsight.com/blog/critical-vulnerability-alert-cve-2026-109618-and-cve-2026-109619-php2shell

[63] Norwegian Defence Research Establishment (FFI). From the US to FFI: Kjeller was first online (15 June 1973).
https://www.ffi.no/en/news/from-the-us-to-ffi-kjeller-was-first-online

[64] E. H. Spafford (1988). The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823.
https://www.cerias.purdue.edu/apps/reports_and_papers/search/index/1515

[65] The New Stack. NASA Programmer Remembers Debugging Lisp in Deep Space.
https://thenewstack.io/nasa-programmer-remembers-debugging-lisp-in-deep-space/

[66] R. Garret (2002). Lisping at JPL.
https://flownet.com/gat/jpl-lisp.html

[67] D. M. Ritchie (1993). The Development of the C Language.
https://www.nokia.com/bell-labs/about/dennis-m-ritchie/chist.html

[68] NSA (November 2022; updated April 2023). Software Memory Safety, Cybersecurity Information Sheet.
https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[69] NSA and CISA (June 2025). Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development (press release).
https://www.nsa.gov/Press-Room/Press-Releases-Statements/Press-Release-View/Article/4223298/

[70] The White House ONCD (26 February 2024). Back to the Building Blocks: A Path Toward Secure and Measurable Software.
https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/memory-safety-fact-sheet

[71] CISA and FBI (January 2025, version 2). Product Security Bad Practices.
https://www.cisa.gov/resources-tools/resources/product-security-bad-practices

[72] CISA (3 November 2021). Binding Operational Directive 22-01: Reducing the Significant Risk of Known Exploited Vulnerabilities.
https://www.cisa.gov/binding-operational-directive-22-01

[73] European Union (2024). Regulation (EU) 2024/2847 (Cyber Resilience Act), Articles 14 and 71.
https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng

[74] 内閣府 (2025). 重要電子計算機に対する不正な行為による被害の防止に関する法律（令和7年法律第42号）の施行について.
https://www.cao.go.jp/cybersecurity/pdf/sekousetsumei.pdf

[75] W3Techs (accessed 25 October 2026). Usage statistics of PHP for websites.
https://w3techs.com/technologies/details/pl-php

[76] DORA / Google Cloud (2021). Accelerate State of DevOps 2021.

https://gitlab.com/gitlab-com/www-gitlab-com/uploads/069ee8e2ee6af463cf0aafcd89eda33e/state-of-devops-2021.pdf

[77] GCC. Common Variable Attributes: cleanup.
https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html

[78] LispWorks. Common Lisp HyperSpec: WITH-OPEN-FILE.
https://www.lispworks.com/documentation/HyperSpec/Body/m_w_open.htm

[79] M. K. McKusick (1999). Twenty Years of Berkeley Unix. In Open Sources, O'Reilly.
https://www.oreilly.com/openbook/opensources/book/kirkmck.html

[80] NIST (December 2025). SP 800-218 Rev. 1 (SSDF Version 1.2), Initial Public Draft.
https://csrc.nist.gov/pubs/sp/800/218/r1/ipd

[81] INTERNET Watch (21 October 2026). FX自動売買専用VPS「お名前.com デスクトップクラウド」、不正アクセス受けサーバー停止.
https://internet.watch.impress.co.jp/docs/news/2149829.html

[82] INTERNET Watch (26 October 2026). GMO、「お名前.com デスクトップクラウド」へのサイバー攻撃で詳報、「能動的サイバー防御」の緊急要請も.
https://internet.watch.impress.co.jp/docs/news/2150991.html

[83] piyokango (23 October 2026, updated 24 October). お名前 .com デスクトップクラウドへの不正アクセスについてまとめてみた. piyolog.
https://piyolog.hatenadiary.jp/entry/2026/10/23/014520

[84] PHP Foundation Discourse, php.announce (26 October 2026). [ANNOUNCE] Updated PHP Windows builds for Brotli security fix.
https://discourse.thephp.foundation/t/announce-updated-php-windows-builds-for-brotli-security-fix/6324

[85] The Hacker News (26 October 2026). PHP Issues Second Windows Security Update in Five Days to Fix Brotli Code Execution Flaw.
https://thehackernews.com/2026/10/php-issues-second-windows-security.html

[86] Microsoft Learn. IIS configuration reference: Adding Handlers <add>.
https://learn.microsoft.com/en-us/iis/configuration/system.webServer/handlers/add

[87] AWS Well-Architected Framework, Security Pillar. Shared responsibility.
https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/shared-responsibility.html

[88] The FreeBSD Project. FreeBSD Handbook, Chapter 17: Jails and Containers.
https://docs.freebsd.org/en/books/handbook/jails/

[89] GMOインターネットグループ (24 October 2026, 11:50). 【重要】お名前.com デスクトップクラウドサービスに対するサイバー攻撃について（第4報）.
https://group.gmo/info/alert/20261024/

[90] GMOインターネットグループ (26 October 2026, 23:10). 【重要】「お名前.com デスクトップクラウド」への不正アクセスに関する現在の対応について（第7報）.
https://group.gmo/info/alert/2026102602/