Vol.2021-DPS-186 No.9
Vol.2021-CSEC-92 No.9
2021/3/15

メモリ破損脆弱性を利用した攻撃に対するRustを用いた対策手法

千脇 貴之　橋本 正樹

概要：現在、主要なシステムプログラムは、C 言語や C++言語を用いて作成されている。C 言語や C++言語のような低レベル言語では、セキュリティの問題として古くからメモリ破損脆弱性が発生しており、作成されたソフトウェアにおいては、メモリ関連の脆弱性が多いと報告されている。メモリ破損脆弱性への対策としては、プログラミング言語としてメモリ安全性機能を備えた、Rust 言語が近年注目をされてきている。本研究では、既存のソフトウェアと Rust 言語を低コストで接続可能とすることで、既存ソフトウェアのメモリアクセスの安全性を高め、メモリ破損脆弱性を利用する攻撃による被害の軽減を目指す。提案手法として、C 言語で書かれたライブラリの一部を Rust 言語で書かれたライブラリを呼び出すように置き換える実装を行う。結果として、安全性では、小さいライブラリ (fgets 等) 単位での書き換えにおいては、C 言語に値を返却する際に引数として渡される情報に依存してしまうため、既存手法の欠点を補うことはできなかった。しかし、関数内でメモリ操作が完了するプログラムでは、Rust 言語で記述したライブラリに置き換えることで、C 言語で起きるメモリ破損脆弱性を防ぐことが可能であることが確認できた。また、大きなオーバヘッド発生しないことも確認した。

Countermeasure method using Rust against attack using memory corruption vulnerability

Abstract: Low level languages such as C and C + +, which are used as main system programs, have a memory corruption vulnerability as a security problem, and it has been reported that there are many memory-related vulnerabilities in the created software. As a countermeasure against memory corruption vulnerability, the Rust language, which has a memory safety function as a programming language, has attracted attention in recent years. In this research, by making it possible to connect existing software and Rust language at low cost, we aim to increase the security of memory access of existing software and to reduce the damage by attacks using memory corruption vulnerability.
As a proposed method, an implementation is carried out to replace a part of a library written in C language with a library written in Rust language. As a result, in the case of rewriting in units of small libraries (fgets, etc.), it is not possible to compensate for the defects of the existing method in terms of safety, because the rewriting depends on information passed as an argument when returning a value to the C language. However, it was confirmed that it was possible to prevent the memory corruption vulnerability which occurs in C language by replacing the program in which the memory operation is completed in the function with the library described in the Rust language. We also confirmed that there was no significant overhead.

1. はじめに

1.1 研究の概要

現在、主要なシステムプログラムは、C 言語や C++言語を用いて作成されている。システムプログラムでは、実行速度やメモリ管理の柔軟性が重要となるため、実行速度が速く、メモリ管理をプログラマが比較的自由に行える C 言語や C++言語が利用されている。一方で、その柔軟さにより、プログラマの責任でメモリ管理を行う必要が生じ、管理が不適切の場合にメモリ破損脆弱性が入りこむ可能性がある。実際のところ、C 言語や C++言語のような低レベル言語では、セキュリティの問題として古くからメモリ破損脆弱性が発生しており、MITER 社が公開した 2019 CWE Top 25 Most Dangerous Software Errors では、上位にメモリ関連のエラーがランクインしている [1]。具体的なソフトウェアについて見ても、Chrome では、2015 年以降

に見つかった脅威度の高い脆弱性の内約 70%がメモリ安全性に関する問題だと報告されている [2] し、MacOS 10.14 でもメモリに関するバグが、見つかった脆弱性の内約 70%であるとされている [3]。また、Android においても、2018 年の重大度の高い脆弱性を分類すると、約 90%がメモリ関連の脆弱性であると報告している [4] し、Microsoft が公開していたパッチは 70%がメモリ関連の修正であったと報告されている [5]。

メモリ破損脆弱性への対策としては、プラットフォームでの対策やランタイムでの対策、プログラミング言語での対策があげられるが、近年、プログラミング言語としてメモリ安全性機能を備えた Rust 言語が注目されている。Rust 言語は、メモリ安全性を備えてながらも、システムプログラミングとして利用できるだけの実行速度と、ある程度柔軟にメモリ操作を行える言語である。すなわち、Rust 言語は、C 言語や C++言語と同様にプログラマが、メモリ内にどのようにデータを配置し、どのタイミングでメモリを解放するかを操作することができる上で、C 言語や C++言語とは異なり、型の安全性とメモリの安全性を保証することができるプログラミング言語である。

本研究は、Rust 言語の持つこれらの特徴を既存のプログラムに適用し、メモリ破損脆弱性への対策とする手法について提案するものである。提案する手法は、既存のプログラムと Rust 言語を無理なく低コストに接続可能とすることを目指し、これにより、既存プログラムのメモリアクセスの安全性を高め、メモリ破損脆弱性を利用する攻撃による被害の軽減を目指す。Rust 言語にその一部を任せることで、プログラマのミスに起因するメモリ破損脆弱性を減らすことができるし、また、C 言語におけるメモリ操作を Rust 言語に置き換えることで、メモリ破損脆弱性が入り込み得る部分を小さくできる可能性がある。Rust 言語を利用するメモリ破損脆弱性対策の検討として、Unsafe Rust や、Rust 言語から C ライブラリ等の他言語の関数を利用する際のメモリ安全性等が近年各所で研究されている。他方で、その課題としては、現状多くのシステムプログラムは依然として C 言語や C++言語での開発が行われており、上記研究成果を適用することを考えた時、様々な既存のプログラムについて、その全体を書き直すことは現実的ではない。そのため、C 言語や C++言語で開発された既存プログラムについて、メモリ安全性に問題や不安がある部分のみに着目し、Rust 言語で書き換えることが試みられている。提案手法においては、C 言語によるプログラムが呼び出すライブラリの一部を、Rust 言語で書かれたライブラリを呼び出すように置き換えることを検討した。Rust 言語を呼び出すように変更する際には、できる限り C プログラマへの負担を減らすために、コンパイル時に動的リンクするライブラリの順番を変更するのみで、Rust のプログラムを実行できるように実装した。提案手法の評価としては、安全性、等価性、有効性、ディスクスペース、オーバヘッドの 5 つの観点からの評価を試みた。

1.2 研究の特徴

本研究の特徴としては、以下のものがあると考える。

- リンクするライブラリを変更することで、元のプログラムを変更せずに、安全な Rust 言語のプログラムを実行することが可能となる。
- 現状では、C 言語のメモリ操作が関数内で完了する場合、安全性を適用できる。
- C 言語に Rust 言語の安全性を適用する際の課題 (C 言語へポインタや配列のリターンが発生する場合に問題となる) がはっきりした。
- C 言語のメモリの正確な情報を利用することで、より安全性を確保できる可能性がある。

1.3 本稿の構成

本論文の構成は以下の通りである。はじめに、第 2 章では、研究の背景として、メモリ破損脆弱性とその対策手法、Rust 言語の概要と安全性の確保手法、関連研究について各々説明する。次に第 3 章では、提案手法として、Rust 言語の持つメモリ安全性を C 言語によるソフトウェアに適用するために、C 言語のメモリ操作関連ライブラリを、Rust で作成したライブラリに置き換える手法について説明する。その後、第 4 章では、実験にて提案手法の安全性やオーバヘッド等の評価を行い、その結果について考察する。最後に、5 章では、本研究の課題を示した後に、本稿をまとめる。

2. 研究の背景

2.1 メモリ破損脆弱性とその対策

メモリ破損に関連する脆弱性は、現在まで主要な脅威の一つであり、様々な攻撃手法によりプログラムの制御をのっとり、特権昇格やプログラムのクラッシュなどを引き起こしている。以下では、メモリ破損脆弱性と主要なメモリ破損への対策手法について延べる。

メモリ破損脆弱性には主に、バッファオーバーフローのように確保した領域を超えてメモリに書き込んでしまう空間的なメモリ破損脆弱性と、解放した後のメモリへのアクセスや初期化前のメモリにアクセスする時間的メモリ破損脆弱性がある

これらのメモリ破損脆弱性を利用した攻撃への対策として、いくつかの手法が提案され、実装されてきている。プラットフォームでの対策技術としては、NX bit と ASLR があり、ランタイム (実行時) の対策技術として、SSP と CFI がある。また、メモリ破損を起こさない、メモリセーフな言語での対策としてガベージコレクタを実装した言語や Rust 言語がある。

ガベージコレクタは、アプリケーションのメモリの割り当てを管理する [6]。ガベージコレクタは、自動でメモリ管理を行うため、プログラマがメモリを管理する必要がなく、use-after-free などのメモリの解放に起因したメモリ破損脆弱性を回避することができる。しかし、ガベージコレクタはパフォーマンスオーバーヘッドが生じることや、自発的なメモリ操作ができない点からシステムプログラムには不向きとされている。Java 言語や C #言語、Go 言語などがガベージコレクトを実装している言語となる。

一方で、Rust 言語ではガーベジコレクタを使用せずにプログラマに負担を強いない方法でメモリ管理を実装している。また一方で、安全性を重視した設計になっており、不正なメモリ領域をさすポインタなどを許さないというメモリ安全性を保証したり、マルチスレッドによる並列実行の際のデータ競合をコンパイル時に排除する仕組みなどを備えている。これらの特徴によって、マルチコア化のプロセッサの性能を発揮し、メモリ関連のセキュリティ脆弱性がない、堅牢なソフトウェアの構築が可能となる。

2.2 Rust 言語の安全性

Rust 言語がメモリセーフを実現する手法として、所有権や借用、ライフタイム、境界チェックなどがある [7]。

所有権は、各値は所有者変数と対応し、いかなる時も所有者は一つとなるという規則を適用するものである。この所有権により、他の変数に値を渡した場合には所有権が移動し、所有権は一つとなることから、渡した元の変数からはアクセスすることができなくなる。ライフタイムは、参照が有効になるスコープのことである。ライフタイムにより、所有者変数が範囲外になると値は、自動的に割り当て解除される。範囲は、ブロックで囲まれた範囲 (スコープ内) に制限される。この所有権とライフタイムにより、所有権がなくなった変数からのアクセスを禁止することで、ダングリングポインタによる use-after-free などを防止することができる。また、Rust では配列などで境界チェックを行う [8]。配列には型情報と長さを格納するため、その情報を検証し配列の範囲外へアクセスすることはできず、プログラムがクラッシュする。この機能により、範囲外参照やバッファオーバーフローなどの対策となる。

Rust 言語には C 言語やハードウェアとのやり取りをするために、Unsafe Rust と呼ばれる Rust 言語のメモリ安全性を強制しない書き方をすることもできる [7]。Rust 言語の外では、安全性を保証することができないため、ハードウェアや FFI(外部関数インターフェース) にて低レベル言語とやり取りする場合に必要となる。Unsafe Rust では、誤ったメモリ操作のコードを書くとメモリ破損が発生してしまう。しかし、Unsafe Rust でもいくつかの安全性チェックは行えるため、ある程度の安全性は保たれる。

2.3 関連研究

現在、Rust 言語を用いてメモリ破損脆弱性を防ぐための研究がなされている。先行研究では、Rust 言語の安全性を調査する研究と、安全でない部分 (Unsafe Rust や FFI) を対策する研究があると考える。以下では、この観点に基づいて既存研究を整理し、説明する。

Rust 言語が安全かどうかを調査した研究がいくつかなされている。

Hui らは、Rust 言語のメモリ安全性がどの程度実現できているのかを調査している [9]。研究では、Rust 言語の既存の CVE にて示されているバグを手動で詳細に分析することで原因を特定している。調査の結果 Rust 言語は、メモリ安全性のリスクを unsafe コードに制限することに成功しており、安全な Rust 言語のみを使用している場合に、Rust 言語がメモリセーフティバグの防止に効果的であると述べている。

Boqin らは、実際の Rust 言語プログラムにおける Unsafe Rust の使用法、メモリ安全性の問題、同時実行バグを調査している [10]。研究では、Unsafe Rust を使用する場合にどのように書かれ、どう記述するべきかや、メモリ安全性の問題が起きる原因を明らかにしている。unsafe を使用する場合には、適切にカプセル化することで、安全性を向上させていると述べられている。また、調査した内容を元に、メモリバグと同時実行バグの検出器の作成も行っている。

これらの研究より、Rust 言語を使用する際の安全性と Unsafe Rust を使用した時の安全性の検証を行うことが可能となる。

また、Rust 言語のメモリ破損対策としていくつかの研究がなされている。

benjamin らは、Rust 言語のプログラムを C 言語ライブラリのコードとデータを別のプロセスに分離するサンドボックス手法である、Sandcrust を提案している [11]。Sandcrust は、コンパイル時に、C 言語ライブラリの API をラップする注釈付き関数を、サンドボックス化されたプロセスで実行されているインスタンスへのリモートプロシージャコール (別のアドレス空間のサブルーチンや手続きを実行するもの) に変換する。Sandcrust は、コンパイラやランタイムへの変更は必要なく、ライブラリの API への簡単なアノテーション (メタデータの付与) のみで実現している。Sandcrust により、C 言語ライブラリを使う際でも、C 言語や C++言語よりも Rust 言語の安全性の利点が残っていると述べている。

Huibo らは、Intel SGX にてエンクレーブ内で実行されているソフトウェアのメモリ破損を防ぐための階層化アプローチである Rust-SGX を提案している [12]。Intel SGX SDK の上に、Rust-SGX レイヤーを構築し、SGX と Rust-SGX を繋ぐ正式に実証されたインターフェースを作成し、メモリ安全性を確保している。Rust-SGX は SGX

の機能を完全に維持し、パフォーマンスのオーバヘッドが低いことが示されている。

これらの研究では、Unsafe Rust や C 言語のプログラムに何らかの Rust 言語で記述した処理を挟むことで、安全にしようと試みている。

2.4 研究の狙い

現在の Rust 言語のメモリ破損脆弱性の対策では、Rust 言語から C 言語ライブラリなどの他言語関数を利用しようする際のメモリ安全性について研究されている。様々な既存研究により、Unsafe なアクセスと safe なアクセスを分離することで安全を保つ技術や、他言語関数 (FFI) とのやり取りにおいて安全性を確保することがある程度可能となっている。また、Rust 言語で開発する際に、他言語のライブラリを安全に利用するための研究も行われている。総じて、近年の研究では、Rust 言語で開発することを前提とした際の安全性を確保することを目的としているものが多く見られる。しかし、現実を見ると、広く社会で利用されている多くのシステムプログラムは、C 言語や C++言語で書かれたものが多いし、新規に開発されるものも依然としてこれらの言語によるものがほとんどである。従って、それら全てを Rust 言語によるものに置き換えることは非常に困難である。

実際のところ、Rust 言語を活用する事例としていくつかのプロジェクトが企業から発表されているが、それらは全てを書き換えるというよりも、C 言語や C++言語などのメモリ安全性やスレッド間の安全性に問題や不安がある部分を Rust 言語で書き換える手法や実装が検討されている。全てを Rust 言語で新たに書き直すよりも、一部の不安な部分を安全にする方が開発者と利用者双方にとって利用コストが低く、他のプログラムから利用する際にも、大きな変更を意識する必要がなくなる。また、Rust 言語は学習コストが高いとされていることからも、安全にするためにシステムプログラムを新たに全て Rust 言語で開発することは、現実的ではないと考える。

そこで本研究では、現在システムプログラムにおいて多く利用されている C 言語や C++言語での開発において、Rust 言語のメモリ安全性を利用することを検討する。利用することができれば、既存の C 言語や C++言語で書かれたプログラムの多くにメモリ安全性を適用可能になるため、一部の操作が安全になるとともに、脆弱性が入り込む範囲 (攻撃領域) が小さくなると考える。また、学習コストや利用コストを小さくするために、開発者が現在のプログラムやコンパイラに大きな変更を加えずに安全性を確保することも検討する。利用する際に、既存のものから大きな変更を要することになると、C 言語や C++言語プログラマのコストとなるからである。これにより、C 言語や C++言語プログラマが Rust 言語を気にせずに安全な実装を行えると考える。

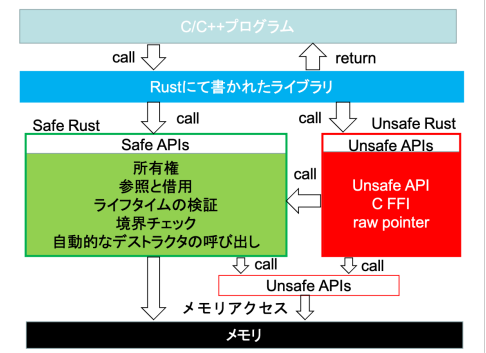

図 1 提案手法の概要図 [9]

3. 提案手法

3.1 提案手法の概要

今回提案する手法は、C 言語で書かれた既存のプログラムに Rust 言語の安全性を適用することを目指すというものである。手法としては、現在ある C 言語のライブラリを Rust 言語で書いた同じ動作を行うプログラムで置き換える。

提案手法では、既存の C 言語/C++言語プログラムで利用されている C 言語ライブラリ呼び出しを、Rust 言語で記述したライブラリを呼び出すように変更し、Rust 言語のプログラムで処理を行うことで、C 言語に Rust 言語の安全性を適用する。Rust 言語では、一部の処理では safe API を利用してメモリアクセスを行えるため C 言語よりも安全に処理を行えると考える。しかし、C 言語との値のやり取りを行う部分などでは、安全性を保証できない Unsafe Rust を使用する必要があるため、実装する際には注意が必要となる。

既存手法からの変更点としては、C 言語で書かれたプログラムを Rust 言語で同じ処理を行うプログラムに置き換えたことである。また、C 言語でコンパイルを行う時に Rust 言語で作成したライブラリを指定するようにコンパイルオプションを変更する。C 言語プログラマは、コンパイルオプションを指定するのみで Rust 言語のメモリ安全性を持つプログラムを利用できる。また、LD_PREROAD を利用することで、再コンパイルなしに Rust 言語のライブラリを指すように変更することも可能である

現状、提案手法における利点としては、C 言語で実装されているプログラムに Rust 言語のメモリ安全性の一部を適用できることだと考える。また、Rust 言語の安全性を適用できる部分においては、安全にメモリ管理を行えるため、C 言語プログラマが気にすべきメモリ管理の負担が軽減されると思われる。実装する際には、ライブラリのリンク時に作成したライブラリのリンク順を先に変更するのみで Rust 言語の関数を呼ぶことができるため、C 言語プロ

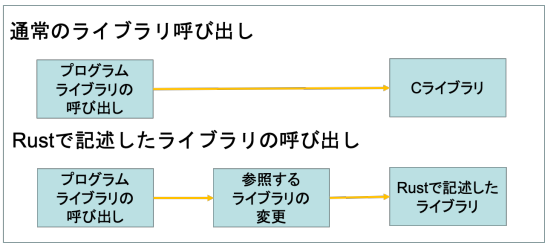

図 2 提案手法と既存手法の実行フローの比較図

グラマのコストも最小限になる。

問題点としては、Rust 言語で処理を行う部分に関しては、型システムやエラーチェックなどが厳格なため、エラーが C 言語のみで記述したプログラムよりも発生する頻度が高くなるのではないかと思われる。C 言語で確保したメモリを操作する際には、メモリ安全性を保証できない Unsafe Rust を使用する必要があるため、実装の際には安全性を十分に検証することが求められる。また、C 言語から Rust 言語で記述されたライブラリを利用するには、ライブラリと同じ動作を行う Rust 言語のプログラムが必要となる。現状では、C 言語からの利用を前提とした C 言語ライブラリと同じ処理を行う Rust 言語のライブラリはほとんど作成されていないため、使用する場合にはプログラムを作成する必要がある。

3.2 提案手法の実装

提案手法の実装について述べる。実装手法としては、既存の C 言語プログラムで利用されているライブラリの関数と同じ処理を行うことのできる Rust 言語のプログラムを作成することである。作成した Rust 言語のプログラムをライブラリファイルとしてコンパイルを行い、置き換えたい C 言語プログラムが作成したライブラリを読み込むようにリンクを行う。リンクを行うことで、Rust 言語のプログラムが関数呼び出しをフックし、実行時に Rust 言語のライブラリにある関数が実行される。フックした関数の処理を Rust 言語のプログラム内にて行い、C 言語に処理を返すことでライブラリの処理を置き換える。Rust 言語では、他の言語と連携する際には、他の言語で関数を利用する場合と同じように記述できるため、C 言語プログラマは特別な記述をほとんどする必要なく、C 言語の関数を呼び出す際と同じ記述方法で利用が可能となる。図 2 に既存手法 (C 言語ライブラリを利用) と提案手法 (作成した Rust 言語ライブラリを利用) の実行フローの比較図を示す。

3.3 実装例

以下で置き換えを実装したライブラリ関数について述べる。今回ライブラリ関数としては、既存の C 言語ライブラリと安全性を確認するために作成したサンプルプログラムの実装を行った。

3.3.1 C 言語ライブラリ

fgets ファイルから読み込み、改行文字または EOF を見つけると、1 行の文字列として切り出し、指定された配列に格納する。最大サイズ-1 に達した場合も、文字列として切り出す。今回は、安全性評価を目的とするため、標準入力の場合のみの実装でファイル処理の部分は未実装 (実装する際は stdin とそれ以外で処理を分ける必要がある。) Rust 言語では、入力処理、最大文字数の判定、戻り値の処理で実装を行った。fgets() では、C 言語から渡される配列の長さ情報があるが、誤った情報が渡された場合に確認する方法がないため、C 言語の際の安全性との違いがあまりないのが現状である。

3.3.2 サンプルプログラム

今回、Rust 言語の安全性を C 言語に適用できることを確かめるために、サンプルプログラムとして以下のものを実装した。今回実装したプログラムは、関数内にてメモリ処理が完結するものである。これらは、libc にある関数ではないため、新たに C 言語で記述したプログラムが必要となる。

配列の境界外アクセス 配列へのアクセスを行う関数。関数内で配列を確保し、インデックスで境界外へアクセスを行う。C 言語では、配列の要素外へのアクセスが発生しても、そのままプログラムの実行を継続してしまい、意図しない動作を引き起こす可能性がある。Rust 言語では、境界外のインデックスアクセスはコンパイルはされるが、実行時に境界チェックによってエラーとしてプログラムを終了させるため、意図しない動作を引き起こさない。

バッファオーバフロー バッファオーバーフロープログラムとして、CTF にて用いられら簡単なプログラムを利用する [13]。コマンドラインからの入力を受け取って、配列に格納するプログラムである。C 言語では配列の境界を超えてアクセスすると、他の変数などを上書きしてしまう。しかし、Rust 言語では入力の際に動的にメモリを確保するため、バッファオーバーフローは発生しない。

ダングリングポインタ ダングリングポインタが発生するプログラムを置き換える。今回は、動的にメモリを確保して、値を代入したものを free したのち、解放した領域を読み出すプログラムとして実装した。C 言語では、free した後にポインタの指す先の値を読み出すとプログラムが実行され、代入した値が出力される。C 言語で free() されたポインタは、そのまま解放されたメモリを挿し続け、値も破棄されるわけではない。Rust 言語では、free ではなく、drop と呼ばれるトレイト (メソッド) によってヒープに確保された値が破棄される。破棄された値を読み込もうとするとコンパイルエラーとなり、実行することはできない。

4. 評価と考察

提案手法として実装したプログラムについて、C 言語の

ライブラリを Rust 言語で置き換えることで、安全なプログラムとなったのか実験して評価する。また、Rust 言語のプログラムで置き換えた際の影響も調査する。実験を行った環境は以下の通りである。

- OS : macOS Catalina ver 10.15.7
- CPU : 2.3Ghz デュアルコア Intel Core i5
- メモリ 16GB 2133MHz LPDDR3
- C 言語コンパイラ : (clang-1100.0.33.17)
- Rust 言語コンパイラ : rustc 1.41.0

評価実験は、安全性・等価性・実行速度・ディスクスペース・汎用性の 5 つの観点から実施する。以下でそれぞれの項目について説明する。

安全性 (書き換えたものが安全か) 安全性の検証手法として、実際にメモリ破損脆弱性を利用した攻撃を試して、攻撃が通らないかどうか実験を行う。

等価性 (書き換えの正しさ) 等価性では、既存の C 言語ライブラリを Rust 言語にて書き換えた際に元の C 言語プログラムと同じ処理を行うかを評価する。評価方法としては、既存の C 言語ライブラリを用いたものと Rust 言語で記述したライブラリをそれぞれ実行し、実行結果が同じとなるかどうかを確かめる。

データ容量 (ディスクスペース) 既存の C 言語ライブラリを置き換えた場合、作成した Rust 言語プログラムにはどれくらい容量が必要となるかを調査する。

実行速度 既存の C 言語ライブラリを用いる場合と Rust 言語のライブラリを用いる場合の実行速度の比較を行う。

汎用性 (有効性) 汎用性として、置き換えるプログラムが実際のプログラム中にてどのくらい使用されているのかなどを、有名な C 言語プログラムを調査して評価を行う。

4.1 評価実験結果

以下にて、評価実験の結果について説明する。実験の際には、C 言語ライブラリの関数と Rust 言語の関数の両方を C 言語から呼び出して実行を行った。

4.2 安全性

作成たプログラムそれぞれについて、実行して安全性の評価を行った。

fgets() gcc のセキュリティオブションにて SSP を無効にしてコンパイルし、C 言語、Rust 言語の両方のプログラムを実行した。C 言語の結果として、 指定した長さ (n の値) が確保した配列の長さを超えた場合にバッファオーバーフローが発生し、他の変数が上書きされた。Rust 言語の結果も、C 言語と同様に、指定した長さ (n の値) が確保した配列の長さを超えた場合にバッファオーバーフローが発生し、宣言していた他の変数が上書きされた。

境界外アクセス 配列を作成して確保したメモリ領域外へアクセスするプログラムを C 言語、Rust 言語それぞれのプログラムで実行した。結果として、C 言語では配列として確保していないメモリ領域を指すプログラムでもエラーにならずに実行された。Rust 言語では、実行時に境界チェックが行われるため、境界外を指すプログラムは実行の際にエラーが発生し、プログラムが終了した。

バッファオーバーフロー 作成したバッファオーバフロープログラムを C 言語、Rust 言語の両方で実行した。結果として、C 言語では、指定したメモリ領域を間違えると、確保した領域を超えて書き込みが発生した。Rust 言語のプログラムでは、入力された値の分のメモリが確保されるため、オーバーフローが発生しなかった。入力に応じて領域が確保されるため、領域を超えて書き込みが発生しないことが確認できた。

ヒープ領域を確保して解放する 作成したヒープメモリ操作プログラムを C 言語、Rust 言語の両方で実行した。結果として、C 言語では free を忘れを防ぐことはできない。Rust 言語では、スコープを抜けると自動でドロップが呼ばれ、値が解放されるため、free 忘れなどが起きない。

ダングリングポインター 作成したダングリングポインターが発生するプログラムを C 言語、Rust 言語の両方で実行した。結果として、C 言語ではダングリングポインタが発生し、解放した変数の値へアクセスを行った場合、解放前に変数に入っていた値が出力された。Rust 言語では、値を drop(破棄) した後に、その変数にアクセスしようとするとコンパイルエラーとなり、プログラムを実行することができなかった。

4.3 等価性

等価性を検証するため、C 言語と Rust 言語それぞれのプログラムを実行し、処理内容の比較を行った。

fgets() 今回は、標準入力の場合の処理を比較した。引数の処理として、C 言語では、ファイルポインタがファイルか標準入力かを判定したが、Rust 言語では場合を分けて記述する必要がある。正常な出力では、引数として与えられたポインタの先頭アドレスが返され、入力文字列が正常に出力された。しかし、Rust 言語では、NULL を扱えないため、失敗の場合に NULL を返却できなかった。

境界外アクセス インデックスが境界内を示す場合には、同じ配列アクセスの操作となり、同じ結果となった。境界外を指す場合には、C 言語では確保した領域を超えてアクセスがなされたが、Rust 言語ではエラーとなった。

バッファオーバーフロー このプログラムは、標準入力から文字列を取得して、メモリ領域に格納する処理である。どちらのプログラムも、確保したメモリサイズ以下であれば、正常に入力処理がなされる。しかし、C 言語での実装では、確保したメモリサイズを超えて入力文字列を書き込

んだ場合に、他の変数の領域を上書きしてしまうことがある。Rust 言語では、入力された文字列に応じたメモリサイズの確保が行われるため、他の変数の領域を上書きすることはない。

ダングリングポインター　プログラムを実行すると、確保した領域に型に応じた値を格納することができる。C 言語では、先にメモリサイズを決定した分の領域が確保されるが、Rust 言語では、値に応じてメモリ領域を確保する。C 言語では、free() を行ったメモリにアクセスし、読み取ると free() 前に格納していた値がそのまま出力される。しかし、Rust 言語では、drop した後にメモリにアクセし、読み取りを行うと向こうなアクセスになり、コンパイルエラーとなる。

4.4　データ容量

今回は、コマンド ls を用いてファイルサイズを計測した。コンパイルでは、動的リンクライブラリを用いるため、C 言語と Rust 言語で実行ファイルのサイズに変化はない。そのため、動的リンクライブラリのファイルサイズを計測した。結果を以下に示す。

表 1　データ容量の比較

ライブラリ関数	C 言語のサイズ	Rust 言語のサイズ
fgets	None	241KB
境界外アクセス	12KB	171KB
バッファオーバーフロー	12KB	207KB
ダングリングポインタ	12KB	261KB

4.5　実行速度

実行速度の比較として、fgets の既存の実装と Rust 言語で置き換えたプログラム、malloc-free を関数化したプログラムの数値と文字列それぞれの場合、バッファオーバーフローが起きるプログラムについて、それぞれ C 言語の実装と Rust 言語で置き換えたものについて計測を行った。malloc-free を関数化したプログラムの実行時間を計測するのは、ダングリングポインタのプログラムは Rust 言語ではエラーとなってしまうので、メモリアロケータを使用した際の C 言語と Rust 言語の実行時間を比較するためである。今回は、置き換えた Rust 言語プログラムを呼ぶ C 言語プログラムと元々の C 言語ライブラリを呼び出すプログラムをそれぞれ 1000 回実行し、その平均タイムを求めた。実験結果を図 3 に示す。

結果として、Rust 言語のプログラムを用いると少しのオーバーヘッドが生じることがわかった。

4.6　汎用性

CTF の問題として fgets() や malloc-free を利用した問題が多く見られた。

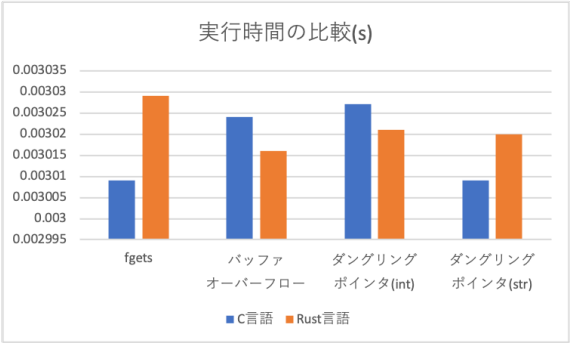

図 3　オーバーヘッドの比較図

4.7　考察

現状の手法では、C 言語に値を渡してしまうと、Rust 言語特有の機能である所有権とライフタイムの保証ができないという問題がある。Rust 言語のメモリアロケータにてメモリを確保した後、C 言語に渡した値を安全に解放するには、C 言語で Rust 言語の関数を呼び出す必要があるため、C 言語で問題となっている free 忘れなどを防ぐことができない。しかし、ライブラリ内でメモリ操作が完結する (malloc-free の対が関数内で呼ばれる) ような関数であれば、保証を提供することが可能となると考える。

バッファオーバフローでは、Rust 言語で処理したものを C 言語に返す場合、確保したメモリの境界情報などは、C 言語から渡される引数に依存するため、安全性を確保するのが難しいという問題がある。そこで、C 言語に値を返す場合には、C 言語の情報に基づく処理を行えば安全に値を返すことができると考える。また、C 言語の情報 (型情報、長さなど) を利用することで、危険な関数を安全にできる可能性がある。長さの情報により、確保したメモリ領域よりも多くの値を返さないようにすることで、バッファオーバーフロー対策ができると考える。

実行時間においては、大きなオーバーヘッドを要さないため、実用上問題はないと考える。等価性においては、Rust 言語で Null を扱う際に問題が生じているので、対応を検討する必要がある。

データ容量では、今回は小さなプログラムを置き換えたため、全体的に小さなファイルとなったが、大きなライブラリを置き換える場合には、それに応じてサイズが大きくなると考える。また、Rust 言語と C 言語のデータ容量を比較すると、明らかに C 言語での実装されたものの方が小さくなるため、使用可能な領域が小さい場合には、問題となる可能性がある。

汎用性においては、CTF という脆弱性を利用した問題においてよく見られるため、これらのライブラリ関数を置き換えることで、安全性を確保できる範囲が広がると考える。

5. まとめと今後の課題

5.1 まとめ

本稿では、メモリ破損脆弱性とその対策技術、Rust 言語の概要と既存の Rust 言語を用いたメモリ安全性の実現手法について述べ、提案手法とその実装について示した。また、実装手法について、安全性の検証やオーバヘッド等の評価を行った。実装や評価によってわかったことは以下通りである。

- Rust 言語から C に値を戻す際など、境界チェックを行うための情報が C の情報に依存するため、C から渡される境界情報などが間違っていると、バッファオーバーフローが起こってしまう。
- Rust 言語内でメモリ操作が完結する場合には、関数を呼び出すようにリンクを変更することで、C 言語で起こるメモリ脆弱性を防ぐことが可能となる。
- 大きなオーバヘッドが発生しないことを確認した。
- 今回の手法では、コストが小さく、プログラマがリンクを変更することで、Rust 言語の安全性を適用できることを確認した。
- Rust 言語で C 言語の操作の一部を置き換えることで、未定義動作を防ぐことが可能となる。

Rust 言語は、メモリ安全性を言語として実装したシステムプログラミング言語であるが、依然としてハードウェア操作や他言語とのやり取りなど Rus の外 (Unsafe Rust) において、メモリ破損脆弱性を起こす危険を持っているため、Rust 言語の外でメモリ安全性を確保する研究が増えると考えられる。また、企業等においても、Rust 言語の利用が検討されて来ているため、今後 Rust 言語は重要になってくると思われる。今後は、C 言語と Rust 言語でのやり取りにおいて互いの情報を保持し、より安全にメモリ操作を行えるように研究を進める。

5.2 今後の課題

今後の課題として、以下があると考える。

- C 言語との Rust 言語で扱いが違ってくるデータへの対処 (NULL) をする。
- C 言語で確保したメモリサイズ知る方法の調査を行う。
- Rust 言語で確保したポインタを C 言語に渡した場合にサイズ情報等の保持が可能か調査を行う。
- 安全性や等価性の証明を行う。

これらを、C 言語での開発を安全にすることを前提に、プログラマへ負うべきコストが小さくなる実装手法を検討していく。また、C 言語に値を戻した際の安全性の問題が残ってしまっているため、C 言語に値を戻す際にも安全性を確保できることを目標とする。今回は一部の C 言語ライブラリにしか適用ができていないため、適用できる範囲を拡大できるように拡張していく必要がある。

参考文献

[1] Miter. 2019 cwe top 25 most dangerous software errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (参照 2020-06-05).
[2] Google. Memory safety - the chromium projects. https://www.chromium.org/Home/chromium-security/memory-safety (参照 2020-08-25).
[3] Languishing. Memory unsafety in apple's operating systems. https://langui.sh/2019/07/23/apple-memory-safety/ (参照 2020-08-25).
[4] Android Security Privacy Team Jeff Vander Stoep and Android Media Team Chong Zhang. Google misc security blog: Queue the hardening enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (参照 2020-08-30).
[5] Microsoft Security Response Center. 2019_01 bluehatil trends, challenge, and shifts in software vulnerability mitigation. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01-BlueHatIL-Trends%2Cchallenge%2Candshiftsinsoftwarevulnerabilitymitigation.pdf: (参照 2020-08-30).
[6] ガベージコレクションの基礎. https://docs.microsoft.com/ja-jp/dotnet/standard/garbage-collection/fundamentals (参照 2020-06-05).
[7] The rust programming language. https://doc.rust-jp.rs/book/second-edition/ (参照 2020-06-05).
[8] C 言語から rust 言語へ. https://sehermitage.web.fc2.com/program/c2rust.html (参照 2020-06-05).
[9] hui xu, zhuangbin chen, mingshen sun, and yangfan zhou. memory-safety challenge considered solved? an empirical study with all rust cves. *pre-print version (preprint). acm, new york*, may 2020.
[10] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. PLDI '20, p. 763–779, 2020.
[11] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pp. 51–57, 2017.
[12] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2333–2350, May 2013.
[13] 清水祐太郎, 竹迫良範, 新穂隼人, 長谷川千広, 廣田一貴, 保要隆明, 美濃圭佑, 三村聡志, 森田浩平, 八木橋優, 渡部裕. セキュリティコンテストのための CTF 問題集. マイナビ出版, Jun 2017.