Security of embedded Systems

Peter Langendörfer

telefon: 0335 5625 350
fax: 0335 5625 671

e-mail: langendoerfer [at] ihp-microelectronics.com
web: http://www.tu-cottbus.de/fakultaet1/de/sicherheit-in-pervasiven-systemen/

Organizational stuff

- Exam: most probably oral
- Lecture schedule subject of change due to „Bahn issues“
- No lectures at:
  - October 22
  - November 5

- Exercises on demand: Dr. Z. Dyka
  - dyka@ihp-microelectronics.com

Outline

- Introduction
- Design Principles
- Memory Management recap
- Attacks
- Protection means
  - Hypervisor/Micro kernels
  - Canaries
  - Isolation
  - Access control/rights management
- Code Attestation
- Secure Code Update

Embedded Systems

Specific Conditions

- *Limited processing power*

  implies that an embedded system typically cannot run applicationsthat are used for defenses against attacks in conventional computer systems (e.g.,virus scanner, intrusion detection system).

- *Limited available power*

  is one of the key constraints in embedded systems. Many such systems operate on batteries and increased power consumption reduces system lifetime (or increases maintenance frequency). Therefore embedded system can dedicate only limitedpower resources to providing system security.

- *Physical exposure*

  is typical of embedded systems that are deployed outside the immediate control of the owner or operator (e.g., public location, customer premise). Thus, embedded systems are inherently vulnerable to attacks that exploit physical proximity of the attacker.

Specific Conditions

- *Remoteness and unmanned operation*

  is necessary for embedded system that are deployed in inaccessible locations (e.g., harsh environment, remote field location). This limitation implies that deploying updates and patches as done with conventional workstations is difficult and has to be automated. Such automated mechanisms provide potential targets for attacks.

- *Network connectivity*

  via wireless or wired access is increasingly common for embedded systems. Such access is necessary for remote control, data collection, updates. In cases where the embedded system is connected to the Internet, vulnerabilities can be exploited remotely from anywhere.

Vulnerabilities

- *Energy drainage (exhaustion attack):*

  Limited battery power in embedded systems makes them vulnerable to attacks that drain this resource. Energy drainage can be achieved by increasing the computational load, reducing sleep cycles, or increasing the use of sensors or other peripherals.

- *Physical intrusion (tampering):*

  The proximity of embedded systems to a potential attacker create vulnerabilities to attacks where physical access to the system is necessary. Examples are power analysis attacks or snooping attacks on the system bus.

- *Network intrusion (malware attack):*

  Networked embedded systems are vulnerable to the same type of remote exploits that are common for workstations and servers. An example is a buffer overflow attacks.

- *Information theft (privacy):*

  Data stored on an embedded system is vulnerable to unauthorized access since the embedded system may be deployed in a hostile environment. Example of data that should be protected are cryptographic keys or electronic currency on smart cards.

Vulnerabilities

- *Introduction of forged information (authenticity):*

  Embedded systems are vulnerable to malicious introduction of incorrect data (either via the system's sensors or by direct write to memory). Examples are wrong video feeds in security cameras or overwriting of measurement data in an electricity meter.

- *Confusing/damaging of sensor or other peripherals:*

  Similar to the introduction of malicious data, embedded systems are vulnerable to attacks that cause incorrect operation of sensors or peripherals. An examples is tampering with the calibration of a sensor.

- *Thermal event (thermal virus or cooling system failure):*

  Embedded systems need to operate within reasonable environmental conditions. Due to the highly exposed operating environment of embedded systems, there is a potential vulnerability to attacks that overheat the system (or cause other environmental damage).

- *Reprogramming of systems for other purposes (stealing):*

  While many embedded systems are general-purpose processing systems, they are often intended to be used for a particular use. These systems are vulnerable to unauthorized reprogramming for other uses. An example is the reprogramming of gaming consoles to run Linux.

Attacks and Countermeasures

Software attacks

- Code injection attacks
  - are examples of software attacks which today comprise the majority of all software attacks. The malicious code can be introduced remotely via the network.
  - often exploit common implementation mistakes in application programs and are often called security vulnerabilities.
  - Some of the attacks include stack-based buffer overflows, heap-based buffer overflows, exploitation of double-free vulnerability, integer errors, and the exploitation of format string vulnerabilities.
- Countermeasures can be divided into nine groups based on:
  - the system component where the proposed countermeasure is implemented
  - the techniques used for the countermeasures

Attacks and Countermeasures

- Groups of countermeasures against software attacks

1. Sandboxing or damage containment approaches
   - popular method for developing confined execution environments based on the principle of least privilege, which could be used to run untrusted programs.
   - limits or reduces the level of access its applications have to the system.

2. Architecture based countermeasures

   return addresses of functions are the most attacked target of buffer overflows, there are many hardware/architecture assisted countermeasures that aim to protect these addresses.
   - Canaries, guard word between return address and local variables
   - ensuring code integrity at runtime.

Attacks and Countermeasures

- Groups of countermeasures against software attacks

3. Operating system based countermeasures
   - Most of the existing operating systems split the process memory into at least two segments, code and data.
   - Marking the code segment read-only and the data segment non-executable

4. Anomaly detection techniques
   - detection compares a profile of all allowed application behavior to actual behavior of the application.
   - Any deviation from the profile will raise a flag as a potential security attack
   - potential to detect several type of attacks, which includes unknown and new attacks on an application code.
   - Negative: can lead to a high rate of false positives.

Attacks and Countermeasures

- Groups of countermeasures against software attacks
  5. Static code analyzers
     - analyze software without actually executing programs built from that software
     - analysis is performed on the source code and in the other cases on some form of the object code.
     - quality of the analysis performed by these tools ranges from those that only consider the behavior of simple statements and declarations, to those that include the complete source code of a program
     - information collected by these analyzers can be used in a range of applications, starting from detecting coding errors to formal methods that mathematically prove program properties

Attacks and Countermeasures

- Groups of countermeasures against software attacks
  6. Dynamic code analyzers
     - source code is instrumented at compile time and then test runs are performed to detect vulnerabilities.
     - dynamic code analysis is more accurate than static analysis but dynamic code checking might miss some errors as they may not fall on the execution path while being analyzed.
  7. Safe languages
     - Safe languages such as Java are capable of preventing some of the implementation vulnerabilities
     - However, everyday programmers are using C and C++ to implement more and more low and high level applications and therefore the need for safe implementation of these languages exists.
     - Safe dialects of C and C++ use techniques such as restriction in memory management to prevent any implementation errors

Attacks and Countermeasures

- Groups of countermeasures against software attacks.

8. Compiler support
   - most convenient place to insert a variety of solutions and countermeasures without changing the languages in which vulnerable programs are written.
   - Protecting the return address in the stack-frame
   - protect program pointers in the code,
   - prevent buffer overflow attacks by checking the boundaries of the buffers when the data is written

9. Library support
   - attempt to prevent vulnerabilities by proposing new string manipulation
   - functions which are less vulnerable or invulnerable to exploitations
   - E.g strings are always NULL terminated.

Attacks and Countermeasures

- Cryptographic attacks

  exploit the weakness in the cryptographic protocol information to perform security attacks, such as breaking into a system by guessing the password.

  - Countermeasures
    - run-time monitors that detect security violations
    - the use of safe proof-carrying code.

Attacks and Countermeasures

- Side channel attacks
  - are known for the ease with which they can be implemented, and for their effectiveness in stealing secret information from the device without leaving a trace
  - Adversaries observe side channels such as
  - power usage, processing time and electro magnetic (EM) emissions while the chip is processing secure transactions.
- Countermeasures

2. Dummy instruction insertion
3. Code/algorithm modification
4. Balancing
5. Others

Classification of Attacks against embedded systems

- What are the main causes of those successful attacks?
- What are the main vulnerabilities?
- What are the commonalities of the attacks?
- How can we use the knowledge to improve the security

Common Vulnerabilities and Exposures (CVE) used as a basis for analysing these questions

- 60,000 records, in total
- 3826 relevant CVE records,
- Dominated by a small number of embedded device manufacturers (e.g., 3306 related to CISCO products).
- 106 CVE analysed

Classification of Attacks against embedded systems: Preconditions

- Internet facing device:

  Many vulnerabilities in the CVE records are potentially exploitable by a remote attacker if the device is connected to the Internet. The attacker dose not necessarily need to have access privileges; the only requirement is that the attacker can potentially discover the device and send messages to it via the network.

- Local or remote access to the device:

  This precondition requires the attacker to have some privileges that allow for logical access to the services or functions provided by the device. This logical access can be restricted to local access or it can be a remote access capability (e.g., via the Internet). Often, the privileges required by the access are normal user privileges, and not administrator privileges.

- Direct physical access to the device:

  Direct physical access requires the attacker to access the device physically. However, the attacker might not need any privileges to access the services of the device.

Classification of Attacks against embedded systems: Preconditions

- Physical proximity of the attacker:

  In some cases, the attacker does not need physical access. It is sufficient that the attacker can be in the physical proximity of the device. For instance, attacks on wireless devices may only require to be within the radio range of the target device.

- Miscellaneous:

  a number of other preconditions was observed in CVE records, each appearing in only one or a few records, An example for a miscellaneous precondition is when the target device has to run some software or has to be configured in a certain way for the vulnerability to be exploitable.

- Unknown:

  Some CVE records and other sources do not provide sufficient amount of information to determine the preconditions of a potential attack; in these cases, the precondition was classified as unknown.

Classification of Attacks against embedded systems: Types of vulnerabilities

- Programming errors:

  Many of the vulnerabilities in the selected CVE records stem from programming errors, which may lead to control flow attacks (e.g., input parsing vulnerabilities leading to buffer overflow problems, and memory management problems such as using pointers referring to memory locations that have been freed).

- Web based vulnerability:

  Many embedded devices have a web based management interface through which they can be configured and updated. However, the web server applications running on those devices are typically rarely updated. Hence, those devices are exposed to web based attacks that exploit unpatched vulnerabilities in the web based interface of the device.

Classification of Attacks against embedded systems: Types of vulnerabilities

- Weak access control or authentication:

  Many devices use default or weak passwords, and some devices have hard-coded passwords that provide backdoor access to those who know the hard-coded password. Such vulnerabilities make it possible for attackers to bypass access control mechanisms rather easily with minimal effort.

- Improper use of cryptography:

  Some devices use cryptographic mechanisms for authentication purposes or for preserving the confidentiality of some sensitive information. Often, cryptographic mechanisms are not used appropriately, which leads to fatal security failures. Examples include the use of weak random number generators for generating cryptographic keys, or vulnerabilities in the protocols that use cryptographic primitives.

- Unknown:

  Similar to precondition, some CVE records do not contain information about the vulnerability itself, while they described the target and the effect of the potential attacks exploiting the unspecified vulnerability

Attack Methods

- Control hijacking attacks:

  This type of attacks divert the normal control flow of the programs running on the embedded device, which typically results in executing code injected by the attacker.

- Reverse engineering:

  Often, an attacker can obtain sensitive information (e.g., an access credential) by analyzing the software (firmware or application) in an embedded device. This process is called reverse engineering. By using reverse engineering techniques, the attacker can find vulnerabilities in the code (e.g., input parsing errors) that may be exploited by other attack methods.

- Malware:

  An attacker can try to infect an embedded device with a malicious software (malware). consequences may go beyond the cyber domain. For instance, the infamous Stuxnet worm reprogrammed PLCs in an uranium enrichment facility, which ultimately led to the physical destruction of the uranium centrifuges controlled by the infected PLCs.

- Injecting crafted packets or input:

  injection of crafted packets is an attack method against protocols used by embedded devices. A similar type of attack is the manipulation of the input to a program running on an embedded device. Both packet and input crafting attacks exploit parsing vulnerabilities in protocol implementations or other programs. In addition, replaying previously observed packets or packet fragments can be considered as a special form of packet crafting, which can be an effective method to cause protocol failures.

Attack Methods

- Eavesdropping:

  While packet crafting is an active attack, eavesdropping (or sniffing) is a passive attack method whereby an attacker only observes the messages sent and received by an embedded device. Those messages may contain sensitive information that is weakly protected or not protected at all by cryptographic means. In addition, eavesdropped information can be used in packet crafting attacks (e.g., in replay type of attacks).

- Brute-force search attacks:

  Weak cryptography and weak authentication methods can be broken by brute force search attacks. Those involve exhaustive key search attacks against cryptographic algorithms such as ciphers and MAC functions, and dictionary attacks against password based authentication schemes. In both cases, bruteforce attacks are feasible only if the search space is sufficiently small. Unfortunately, CVE records report such vulnerabilities.

- Normal use:

  This refers to the attack that exploit an unprotected device or protocol through normal usage. Some CVEs reported on potential attacks where the attacker simply used some unprotected mechanism as if he was a legitimate user. For instance, the attacker can access files on an embedded device just like any other user if the device does not have any access control mechanism implemented on it.

- Unknown:

  some CVEs described vulnerabilities but did not identify any particular attack method that would exploit those identified vulnerabilities.

effect of the attacks

- Denial-of-Service:

  Many CVE records identify potential attacks that lead to denial-of-service conditions such as malfunctioning or completely halting the device.

- Code execution:

  Another large part of the analyzed CVE records identify execution of attacker supplied code on the embedded device as the effect of potential attacks. This also includes web scripts and SQL injections, not only native code of the device.

- Integrity violation:

  A commonly observable effect of potential attacks is the integrity violation of some data or code on the device. This includes modification of files and configuration settings, as well as the illegitimate update of the firmware or some applications on the device.

- Information leakage:

  In some cases, the effect of the attack is the leakage of some information that should not be obtained by the attacker.

Effect of the attacks

- Illegitimate access:

  Many attacks result in the attacker gaining illegitimate access to the device. This not only includes the cases when an attacker, who otherwise has no access to the device, manages to logically break into it, but also cases when the attacker has already some access, but he gains more privileges (i.e., privilege escalation).

- Financial loss:

  Certain attacks enable the attacker to cause financial loss to the victim e.g. by making calls from a smart phone. Meant here only those attacks whose primary goal is to cause financial loss.

Effect of the attacks

- Degraded level of protection:

  In some CVE records, the potential attack results in a lower level of protection than expected. An example would be when a device is tricked into using weaker algorithms or security policies than those that it actually supports.

- Miscellaneous:

  Some attacks cause users to be redirected to malicious websites or traffic to be redirected. In these cases, there is not enough information about what happens exactly to the redirected user or traffic, but there is information about the effect.

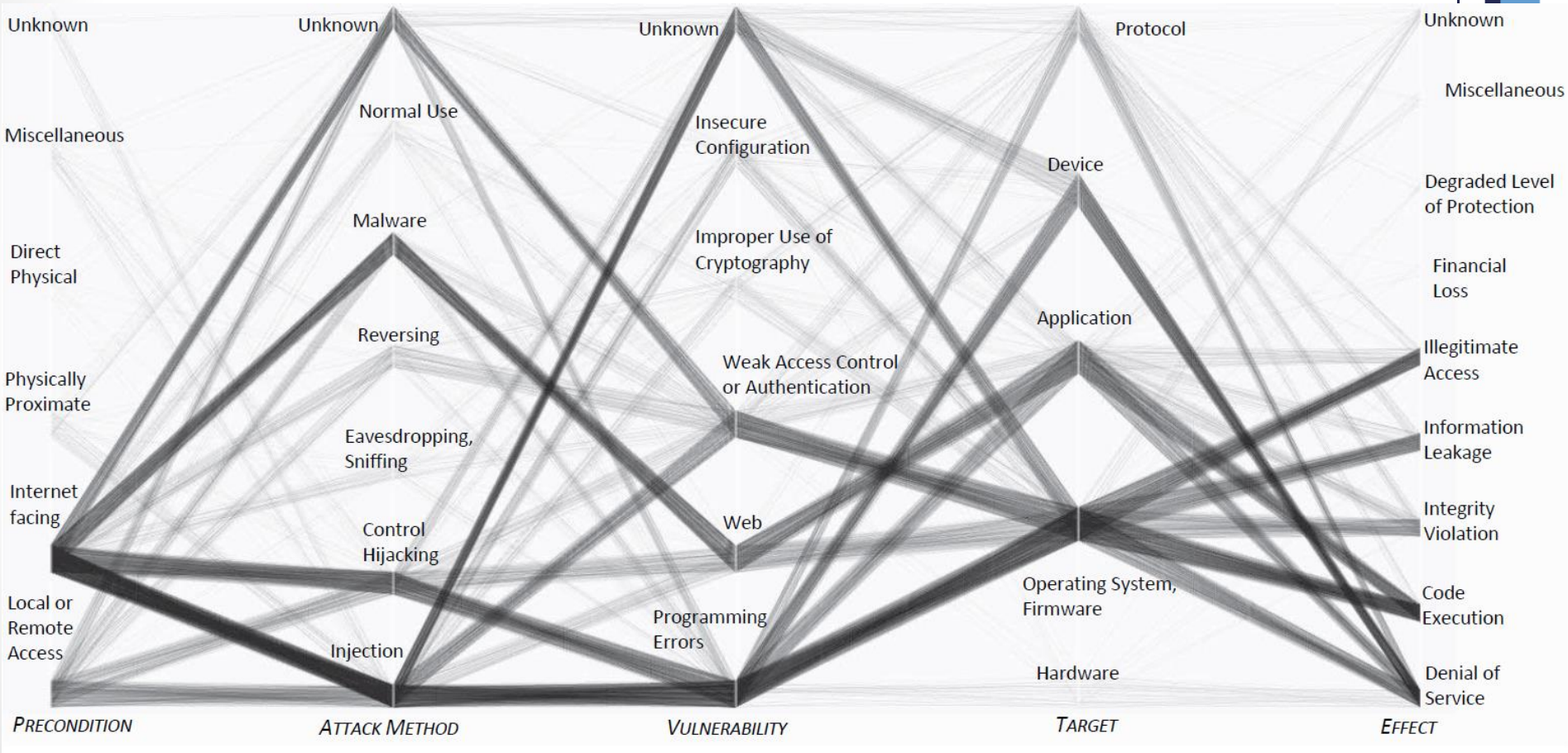
- Unknown:

  In some CVE records, no specific attack effect is identified. This is mainly the case when the attack method is not identified either, and the CVE record contains only information about a vulnerability.

Common Attack Scenarios

- most of the time the attacker will either
  - inject crafted inputs and arguments, or
  - perform a control hijack by exploiting buffer overflows or
  - embedding commands into parameters.
- common exploitation method is the use of malware,
  - when the attacker injects scripts into web pages or
  - is able to install a compromised firmware.
- numerous CVE entries do not state how the vulnerability could be exploited.
- common vulnerabilities:
  - programming errors,
  - web-based vulnerabilities
  - and weak access control or
  - authentication.

Common Attack Scenarios

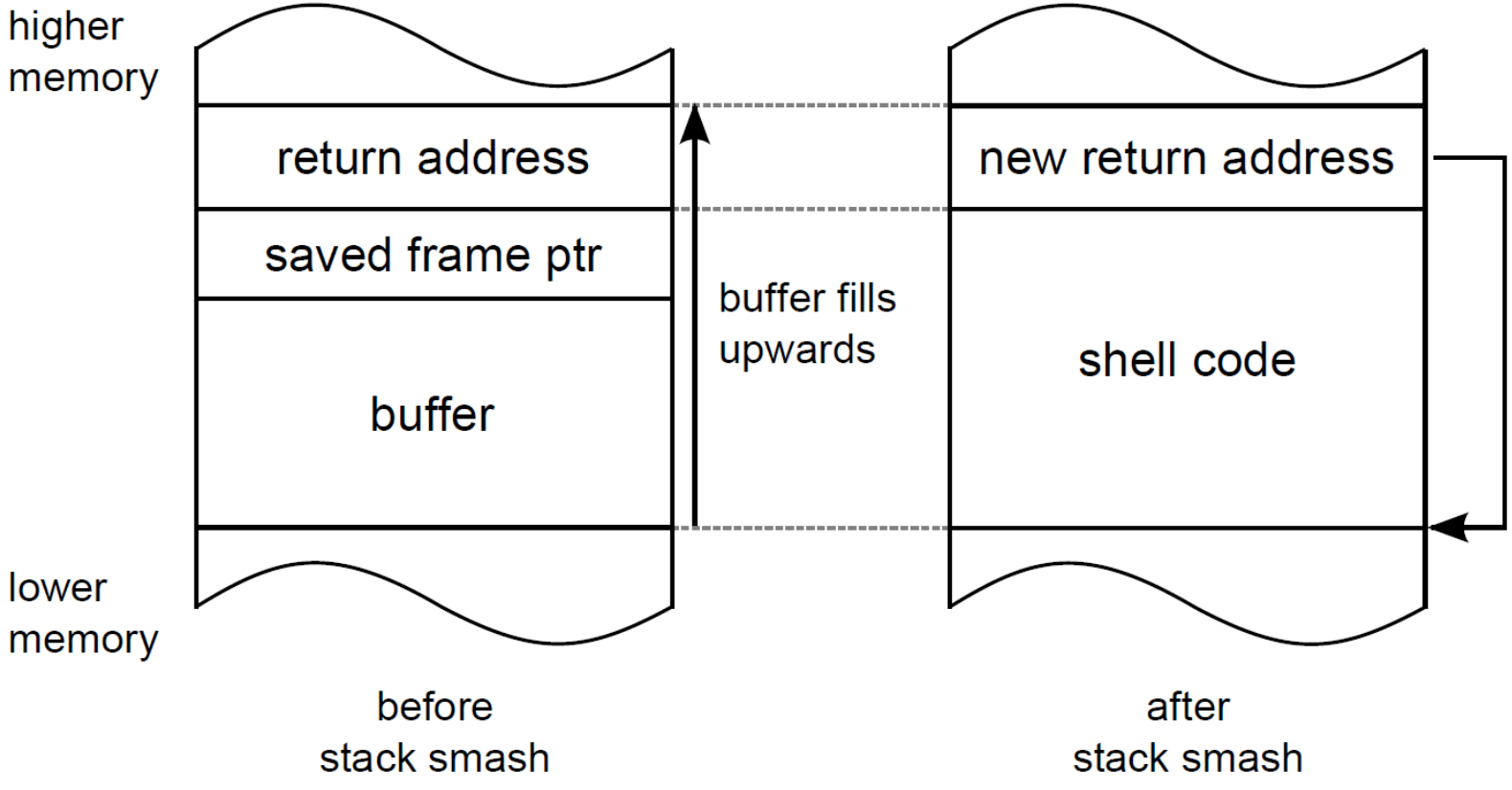


Attack details

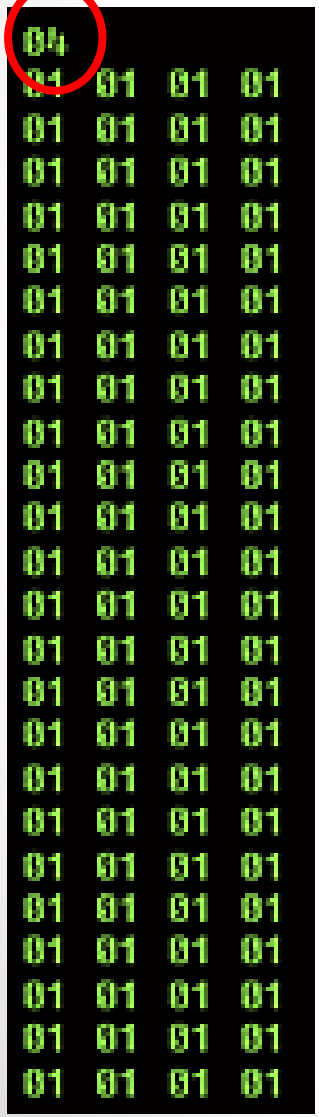
Stack smashing

- most common form of security vulnerability on C implementations
- corrupts the execution stack by writing beyond the end of a local array.
- causes a return from the routine to any address
- stack usually grows down from high to low addresses.
- lower end of the stack is stored in the stack pointer, usually a processor register.
- In case of a subroutine call the current instruction pointer is pushed onto the stack, it is the program address where the program goes on after the subroutine call is finished.

Stack smashing

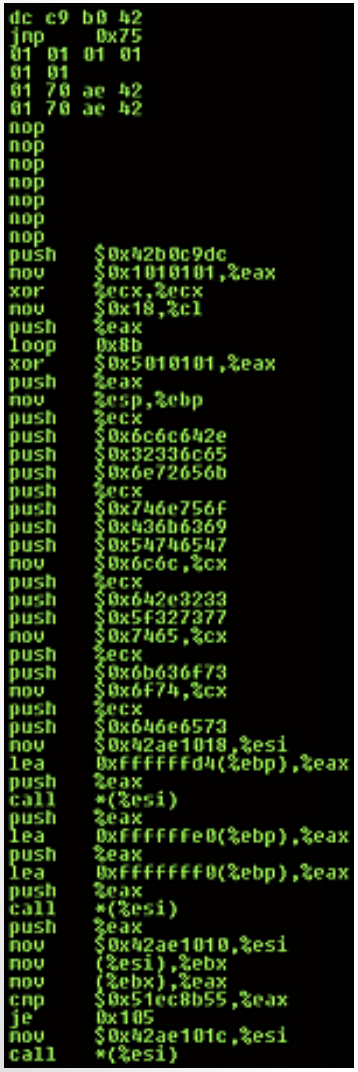


Slammer: *Step 1*



- *Get Inside*
  - Slammer masquerades as a single UDP packet.
  - first byte in the string – 04 – tells SQL Server that the data following it is the name of the online database being sought.
  - Microsoft's tech specs dictate that this name be at most 16 bytes long and end in a telltale 00.
  - But in the Slammer packet, there is no 00
  - SQL software pastes the whole thing into memory.
  - The initial string of 01 characters spills past the 128 bytes of memory reserved for the SQL Server request and into the computer's stack next door

Slammer : *Step 2 and Step 3*



- *Reprogram the Machine*
  - The first thing the computer does after opening Slammer's too-long UDP "request" is overwrite its own stack
  - new instructions here Slammer are put on top of the stack
  - The computer reprograms itself without realizing it.
- *Choose Victims at Random*
  - Slammer generates a random IP address,
  - Slammer looks up the number of milliseconds that have elapsed on the CPU's system clock since it was booted and interprets the number as an IP address.

Slammer : *Step 4 and Step 5*

```
mov     0xffffffb4(%ebp),%eax
lea     (%eax,%eax,2),%ecx
lea     (%eax,%ecx,4),%edx
shl     $0x4,%edx
add     %eax,%edx
shl     $0x8,%edx
sub     %eax,%edx
lea     (%eax,%edx,4),%eax
add     %ebx,%eax
mov     %eax,0xffffffb4(%ebp)
push    $0x10
lea     0xffffffb0(%ebp),%eax
push    %eax
xor     %ecx,%ecx
push    %ecx
xor     $0x178,%cx
push    %ecx
lea     0x8(%ebp),%eax
push    %eax
mov     0xffffffac(%ebp),%eax
push    %eax
```

- *Replicate*
  - The envelope is addressed, IP address generated in Step 3
  - Slammer points to its own code as the data to send.
  - The infected computer writes out a new copy of the worm

```
call    *%esi
jmp     0x142
```

- *Repeat*
  - After sending off the first packet, Slammer loops around
  - Sending another packet to a different computer.
  - Instead of making another call to the system clock, it just shuffles the bits of the IP address

Heap smashing

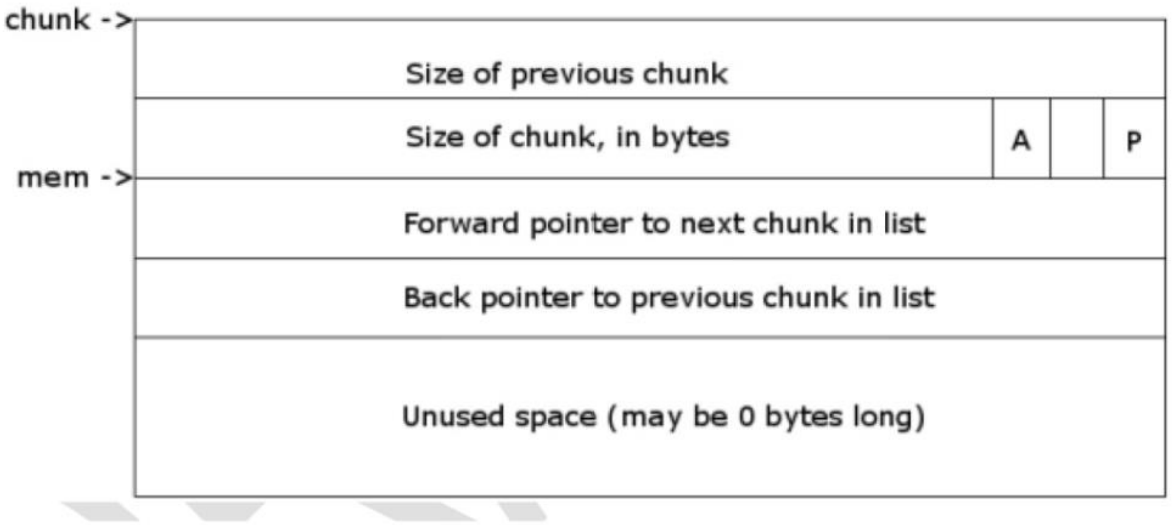
- buffers on the heap are dynamic.
- developer or the compiler does not know which data will be saved there;
- the program is therefore designed to adjust the space needed whilst the program is running. arrays and instances of objects.
-
- “alloc” functions reserve memory in the virtual address space of the process and then returns an address to the process for that allocated memory.
- Needs to be released by the programmer when the time comes.

- Typically, the operating system gives every process a heap of a specific size, although this size can be changed while the program is running.

- If the limits of the memory space provided by the kernel are exceeded, an access violation occurs, and this generally terminates the process.
- there are no such limits within this memory space, and neither the CPU nor the kernel can recognize whether an overflow has occurred within the heap.

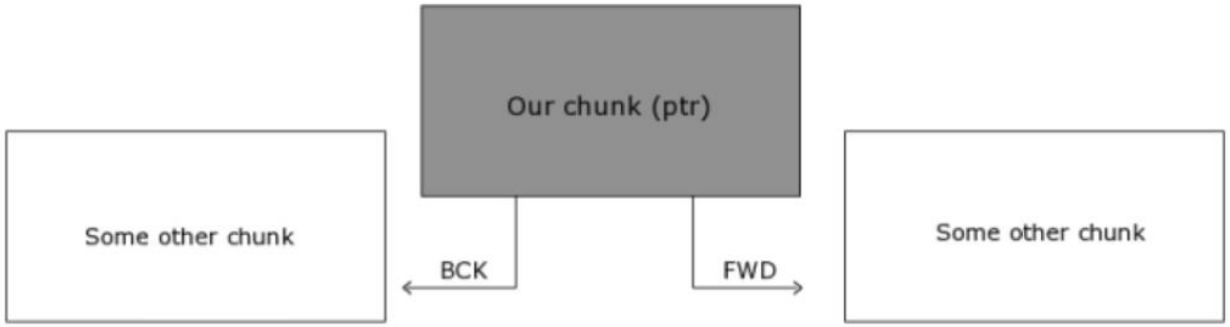
Heap smashing: Double-Free Vulnerabilities

- Freeing the same chunk of memory twice, without it being reallocated in between
- simple case:
  - chunk to be freed is isolated in memory
  - bin (double-linked list) into which the chunk will be placed is empty

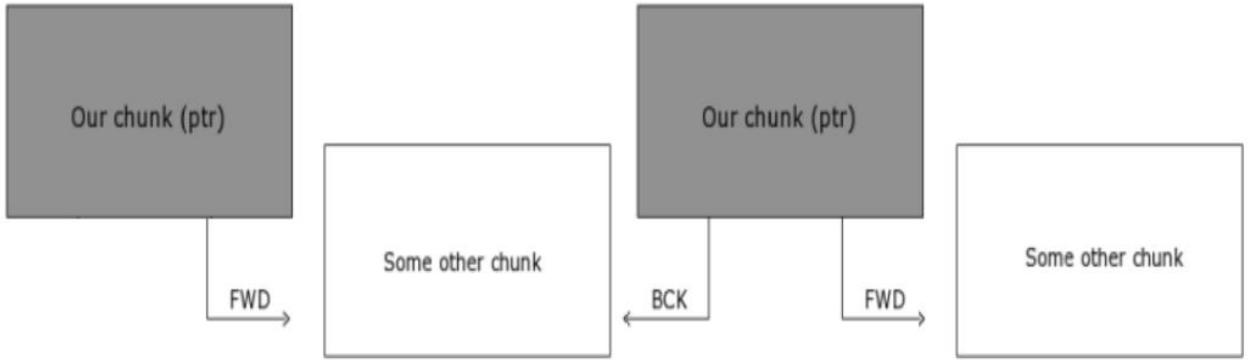


Double-Free Vulnerabilities

- After first call to free



- After 2nd call to free

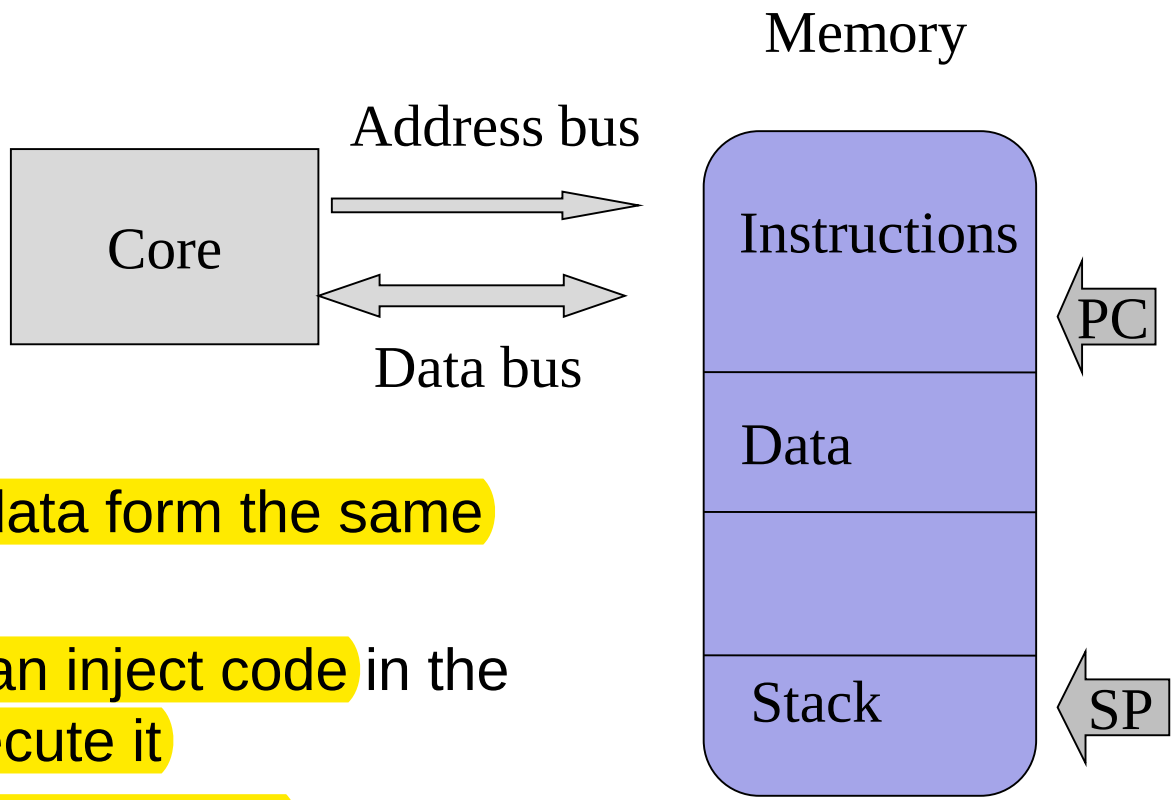


Code Injection Attacks on Harvard-Based Architecture Devices

Standard Code Injection Technique

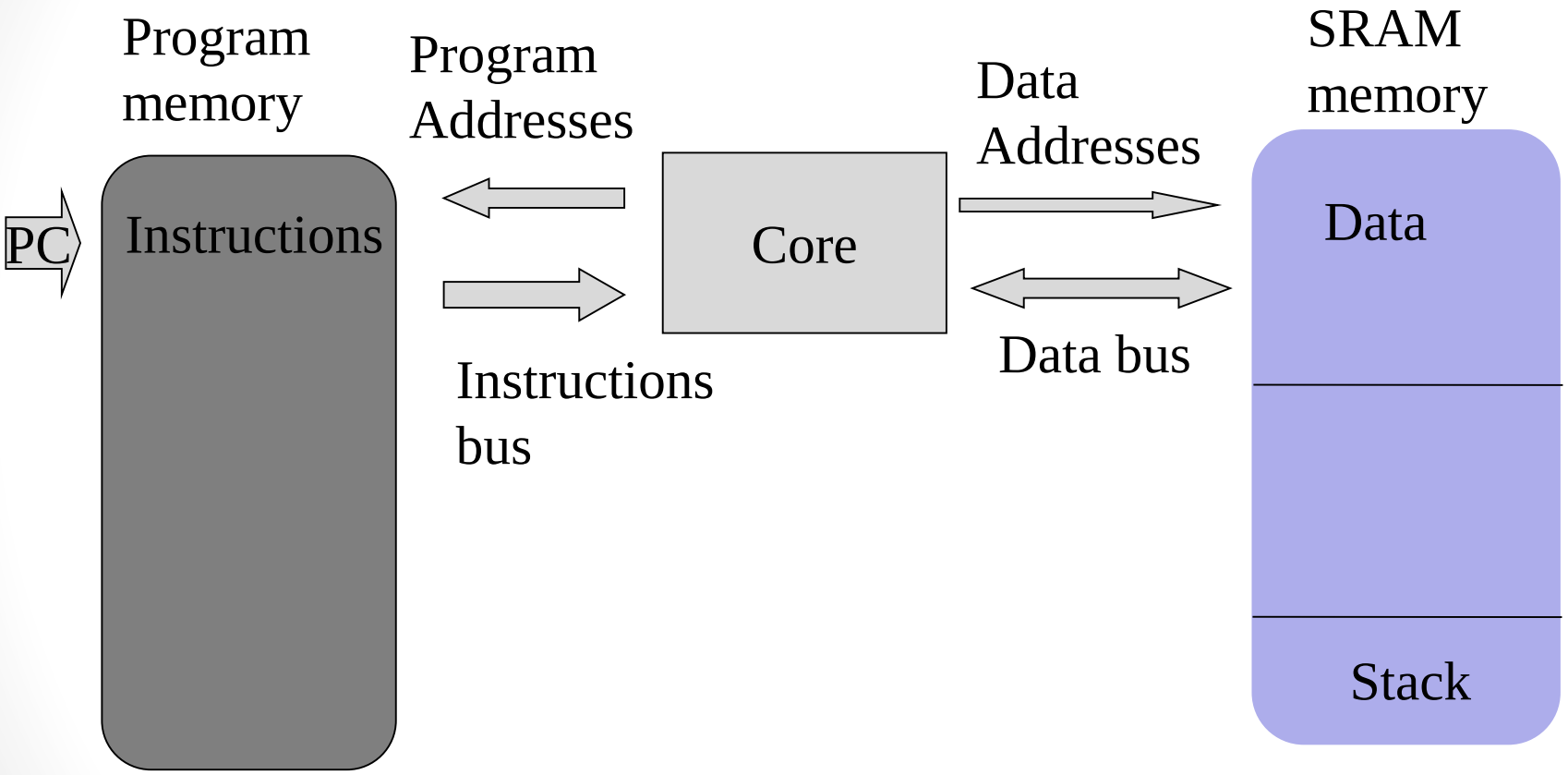
- Standard Code injection technique
  - Overflow a buffer allocated on stack
  - Overwrite return address
  - Inject instructions on the stack
  - Return to those instructions
- Possible on Von Neumann architecture
  - Can be prevented by making Stack Memory non executable (NX, randomizing the address layout...)
  - Not possible on Harvard Architectures
- But Return Oriented Programming still enables to perform some « actions » (more details to come)
  - It's less powerful than code injection

Von Neumann Architecture



- Instructions, data form the same memory
- An attacker can inject code in the stack and execute it
- The most common CPU architecture

Harvard Architecture

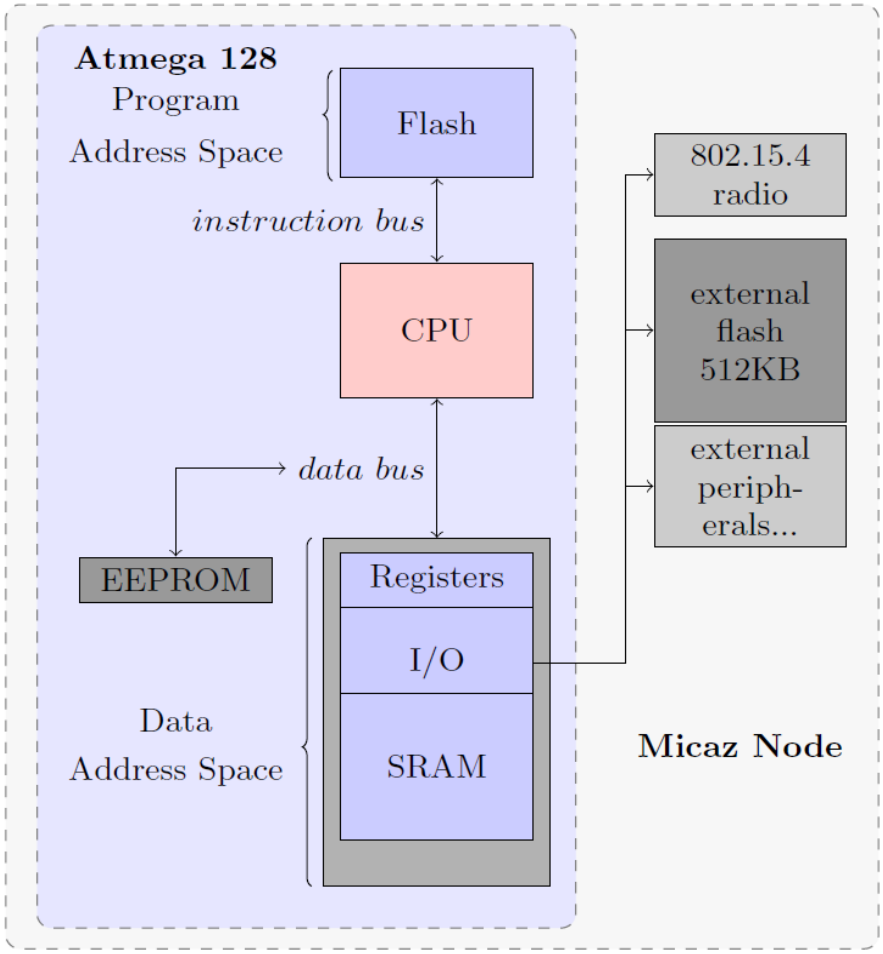


- Instructions, data from separate memory
- An attacker can inject code in the stack but **cannot** execute it
- Common in embedded systems (AVR, PIC) and DSP

Harvard Architecture

- It has been a common belief that code injection into a Harvard architecture is impossible!!…
- We show that this is not only possible but that code injection can also be performed on Harvard-Architecture constrained devices
- We were able to inject code into a Micaz WSN node
  - Constrained memory 4KB data 128KBytes program memory
  - Small network packets (28Bytes) on a IEEE 802.15.4 radio
  - Harvard Architecture (**ATMEL AVR** Atmega128)

Mica-Z Architecture



Attack Assumptions and Building Blocks

- Objective: perform remote code injection on a Micaz
- Assumptions
  - A traditional stack based buffer overflow exists
  - Program Memory contents are known
  - A boot loader is present
- Building blocks :
  - Return Oriented Programming
    - To overcome the Harvard architecture
  - Bootloader
  - Fake stack injection
    - To overcome the packet and stack size limitation

Return Oriented Programming (ROP)

- Generalization of "*return to libc*" attacks
  - introduced by H. Shacham at CCS 2007 on the x86
- Useful when Code cannot be executed in the stack..
- But the control flow can be modified
  - Executes code already present in the program memory
  - By chaining sequences of instructions terminated by a return
    - Called Gadgets
  - Gadgets can be used to perform any action
    - When a Turing Complete Gadget set is available

ROP on Embedded Systems

- ROP is not practical on embedded systems
  - Packets and stack size limitations
  - Code size limitation (gadgets availability)
  - ⇨ Finding a *Turing Complete* Gadget set is unlikely
- But is an useful tool to perform code injection

Bootloader : a key element of the attack

- *Bootloader* is the program used for code update
- *Code update* is a must… otherwise
  - A device with a bug is “dead”
  - Think about pacemakers, sensors in cars ...
- The bootloader consists in :
  - Getting the image (from serial port , local storage , network …)
  - Copying the image to program memory
  - Uses dedicated instructions to copy a page from data to program memory
- We assume that if *remote* code update is present images are authenticated
  - *Otherwise attack is trivial*

Injecting code using ROP on sensors

- To inject code into a device, the attacker has to send a specially-crafted packet that contains:
  - Addresses of the gadgets
  - Gadgets' parameters
  - The malware code to be injected
  - Padding
- I.e. 305 bytes if malware is 256 bytes long

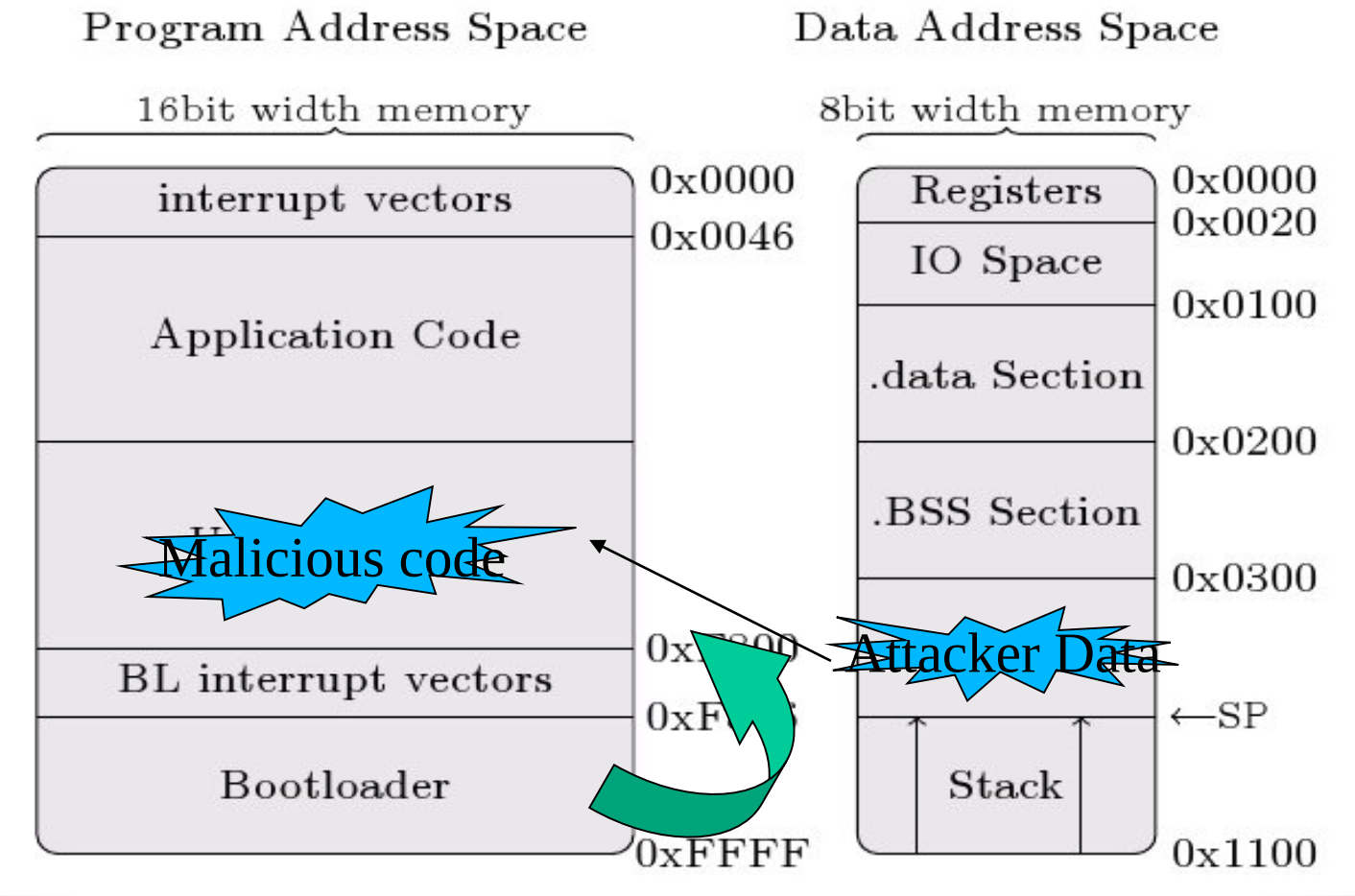
- But this is not possible on Sensors…
- Packet and stack sizes are limited,
  - TinyOS packet payload maximum size is 28 bytes!
  - The number of gadgets that can be linked is limited

Fake stack injection

- Instead of sending one large packet with the large stack in it we send several small specially-crafted packets
  - To build a *fake stack*
- Each packet:
  - Contains one byte of the fake stack
  - Triggers a buffer overflow
  - Copies its fake stack byte into unused data memory, using ROP
- A final packet
  - Triggers a buffer overflow
  - Changes the Stack Pointer to the fake stack, using ROP
  - Executes several gadgets that copy the malware (contained in the fake stack) into the program memory

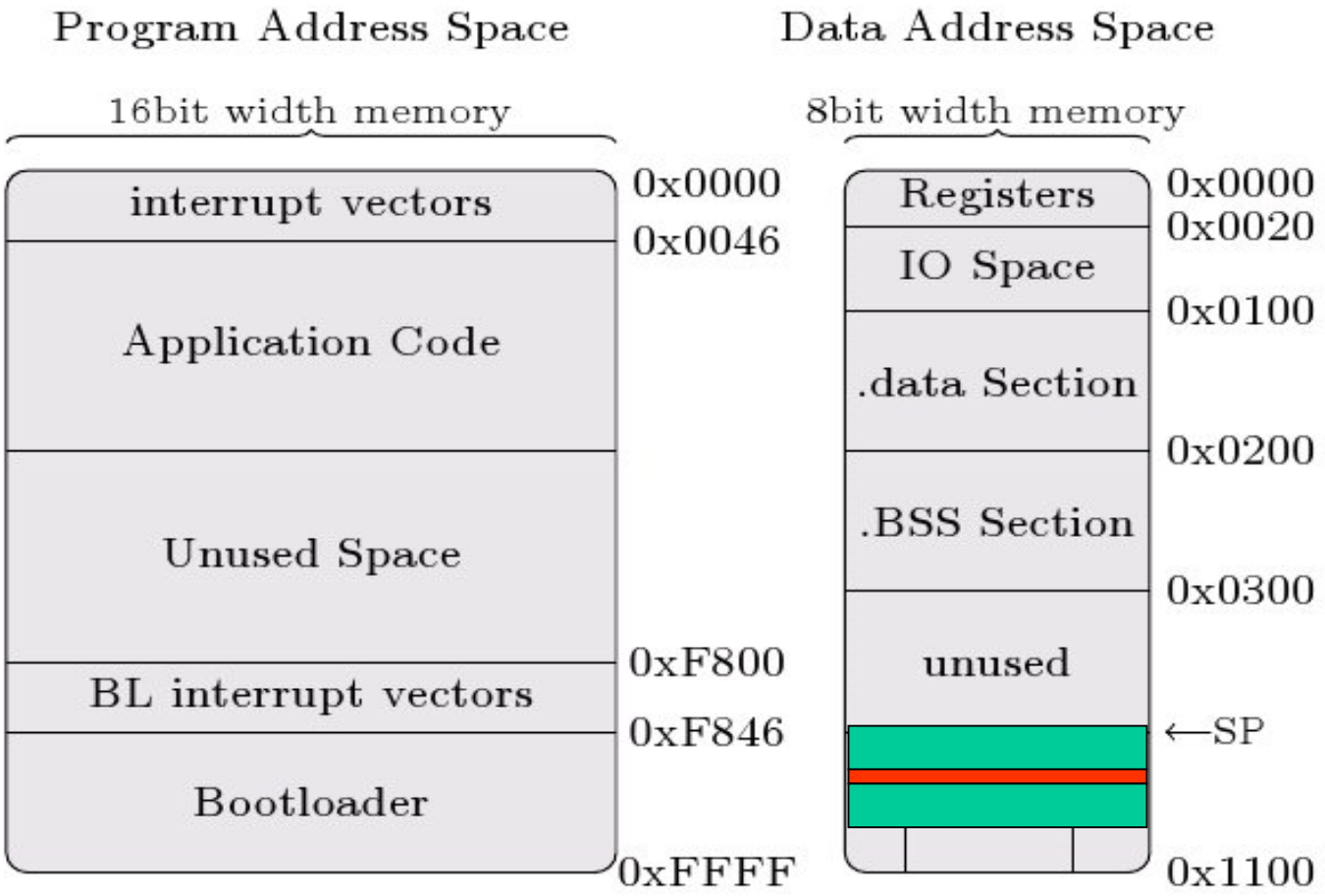
Attack detailed overview

- Injecting code in persistent flash memory



Illustration

- A specially-crafted packet (that contains the first byte of the fake stack) is sent



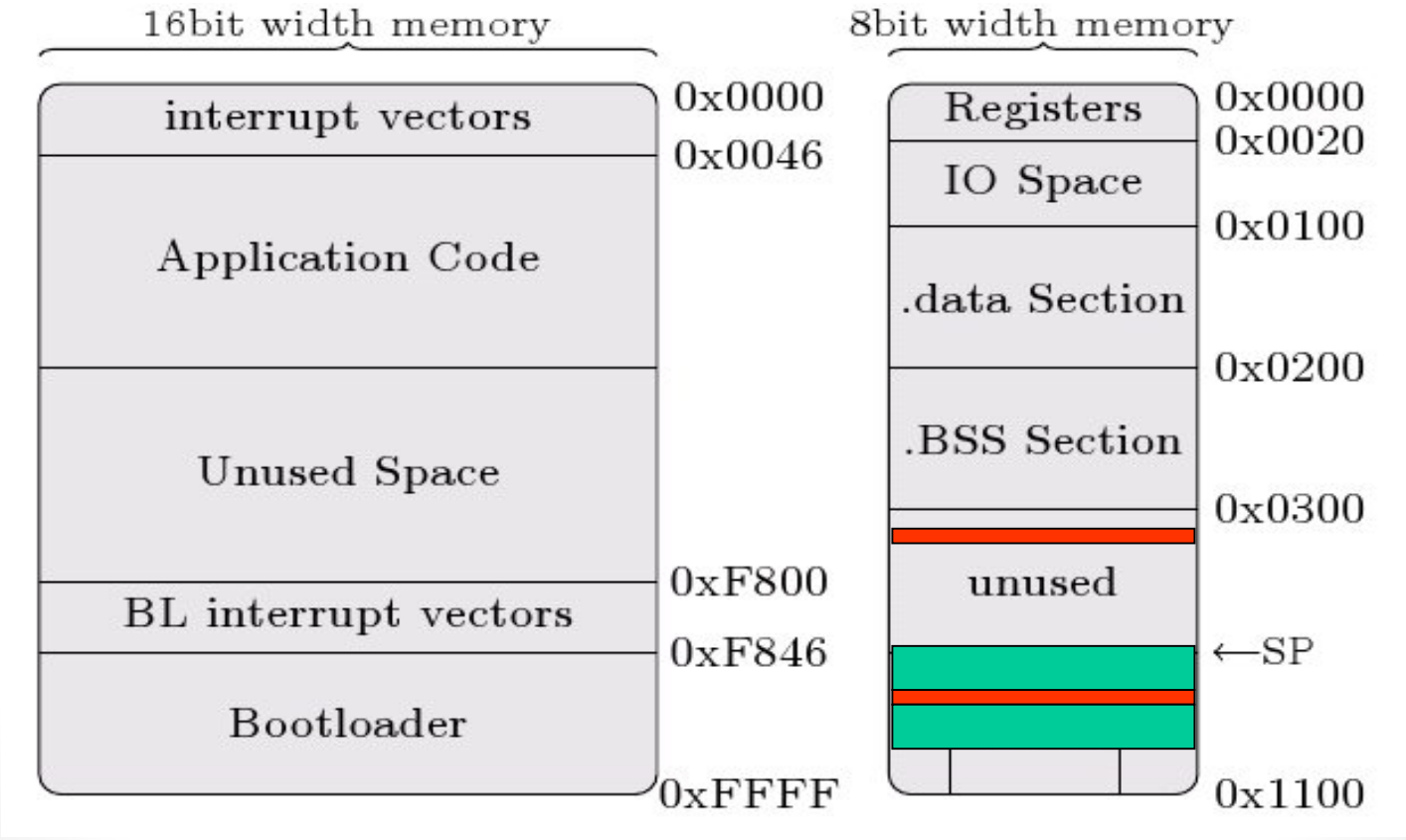
The special packet and the stack

```
uint8_t payload[ ]={
 0x00,0x01,0x02,0x03, // padding
 0x58,0x2b,           // Address of gadget 1
 ADDR_L,ADDR_H,       // address to write
 0x00,                // Padding
 DATA,                // data to write
 0x00,0x00,0x00,      // padding
 0x85,0x01,           // address of gadget 2
 0x3a,0x07,           // address of gadget 3
 0x00,0x00            // Soft reboot address
 };
```

Memory address	Usage	normal value	value after overflow
0x10FF	End Mem		
⋮	⋮	⋮	⋮
0x1062	other	0xXX	$ADDR_H$
0x1061	other	0xXX	$ADDR_L$
0x1060	$@ret_H$	0x38	0x2b
0x105F	$@ret_L$	0x22	0x58
0x105E	tmpbuff[3]	0	0x03
0x105D	tmpbuff[2]	0	0x02
0x105C	tmpbuff[1]	0	0x01
0x105B	tmpbuff[0]	0	0x00

Illustration (2)

- This packet triggers a buffer overflow that copies its fake stack byte into the RAM memory
- And then reboots the node to restore consistent state



Writing a byte into memory (simplified)

- A gadget chain

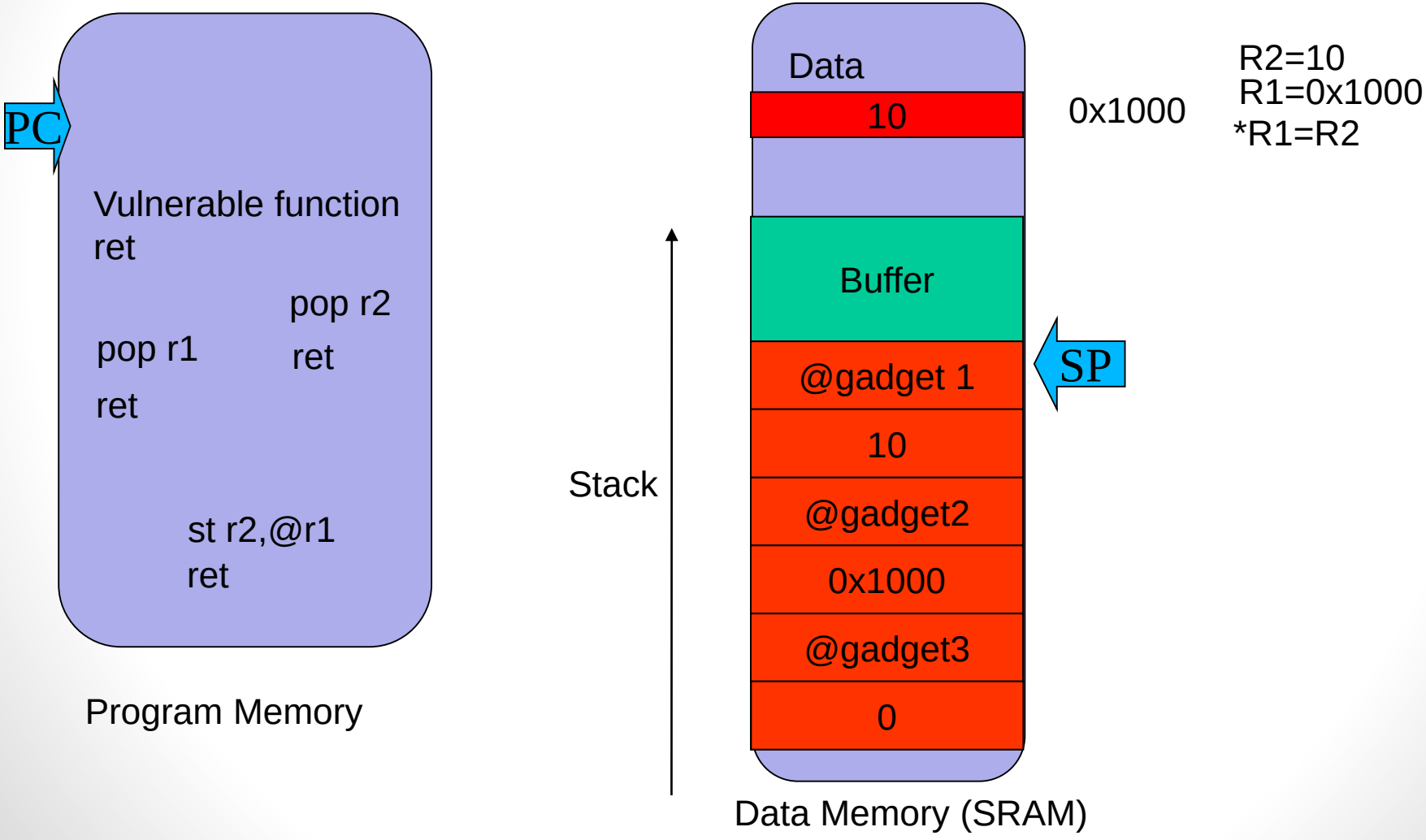


Illustration (3)

- A specially-crafted packet (that contains the second byte of the fake stack) is sent

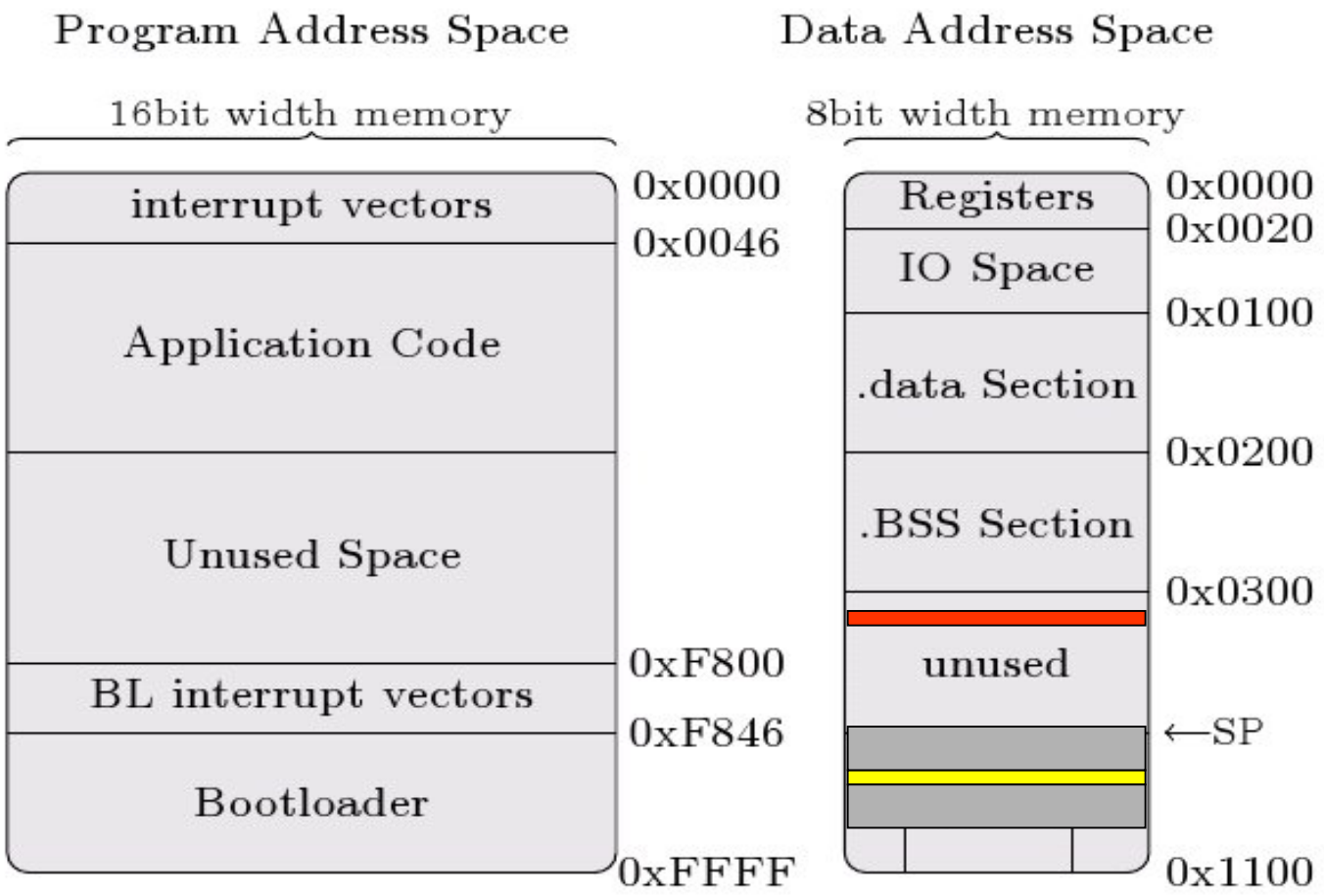


Illustration (4)

- This packet triggers a buffer overflow that copies its fake stack byte into the RAM memory… and reboots the node

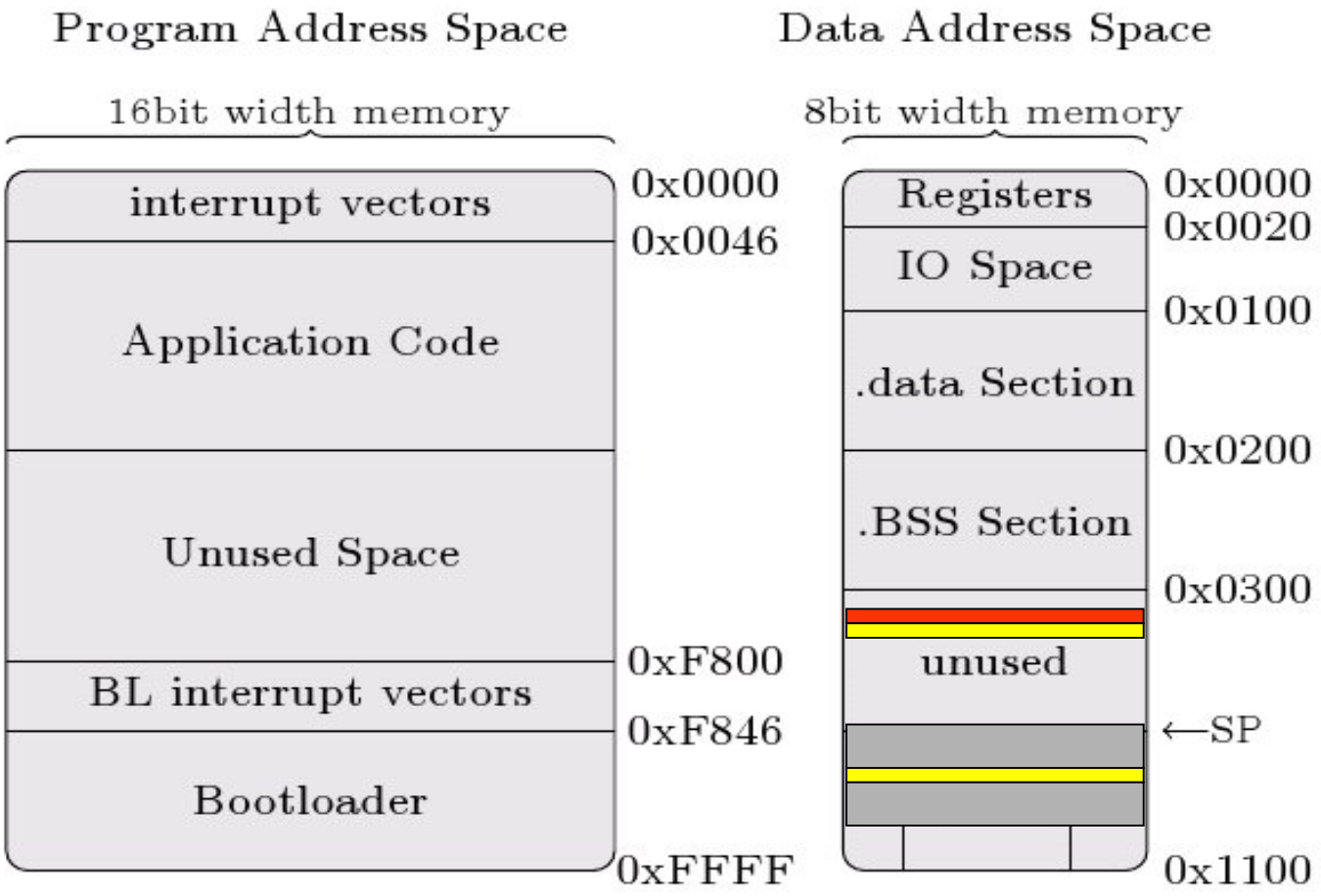


Illustration (5)

- After N packets the fake stack is in memory!

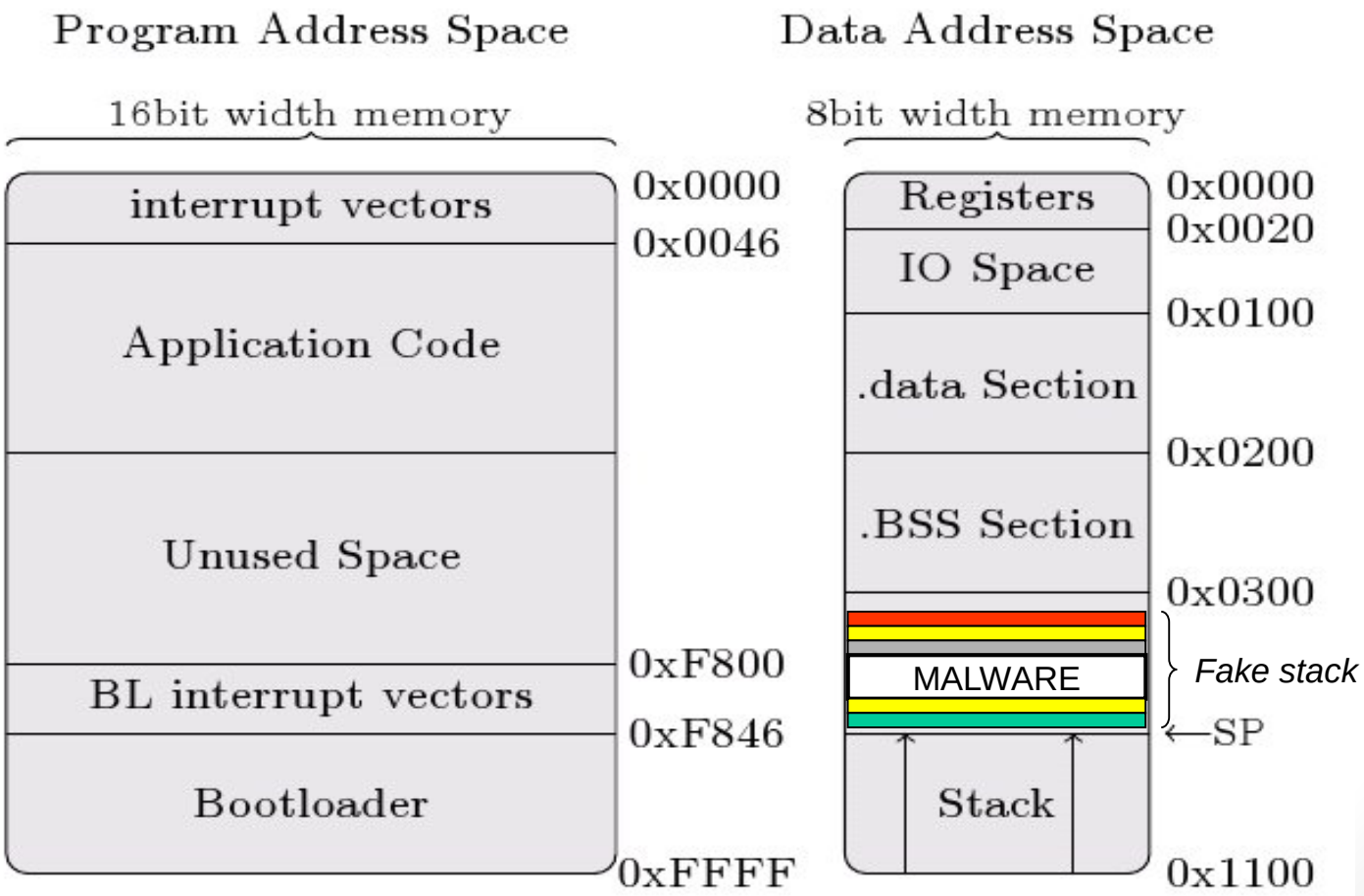


Illustration (6)

- Final a specially-crafted packet is sent…

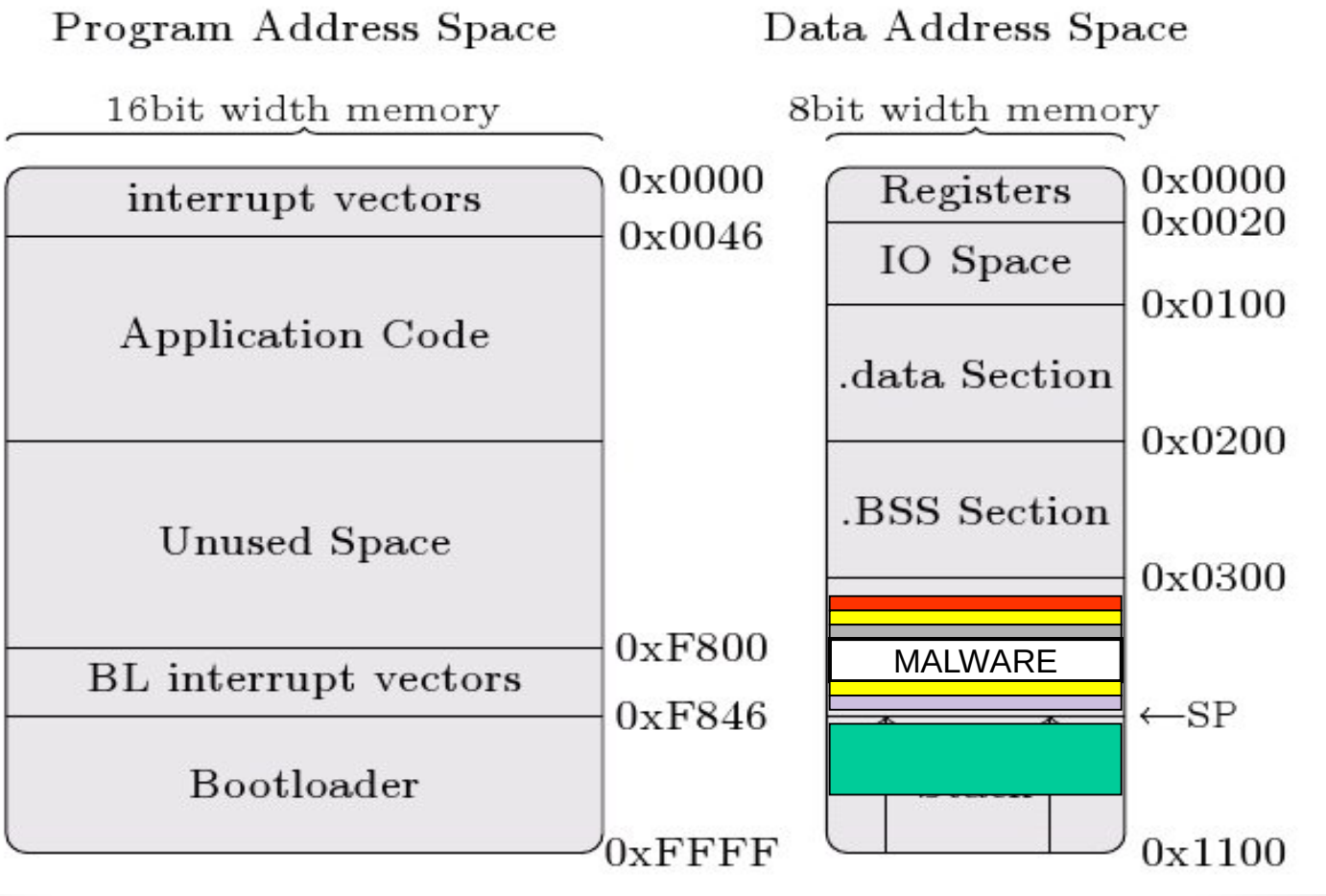


Illustration (7)

- …that triggers a buffer overflow and changes the SP to the fake stack

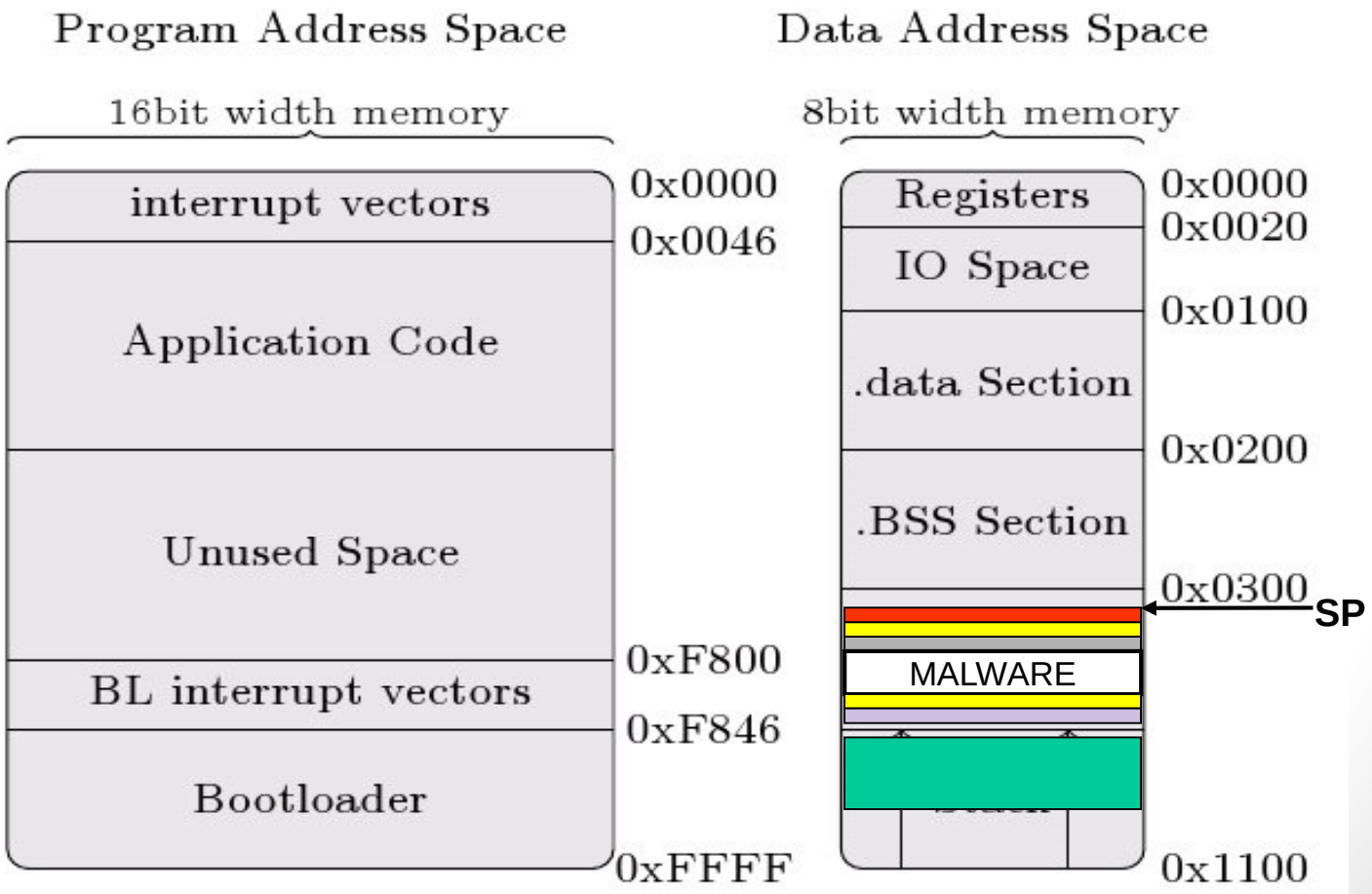


Illustration (8)

- The fake stack is then "executed" / interpreted

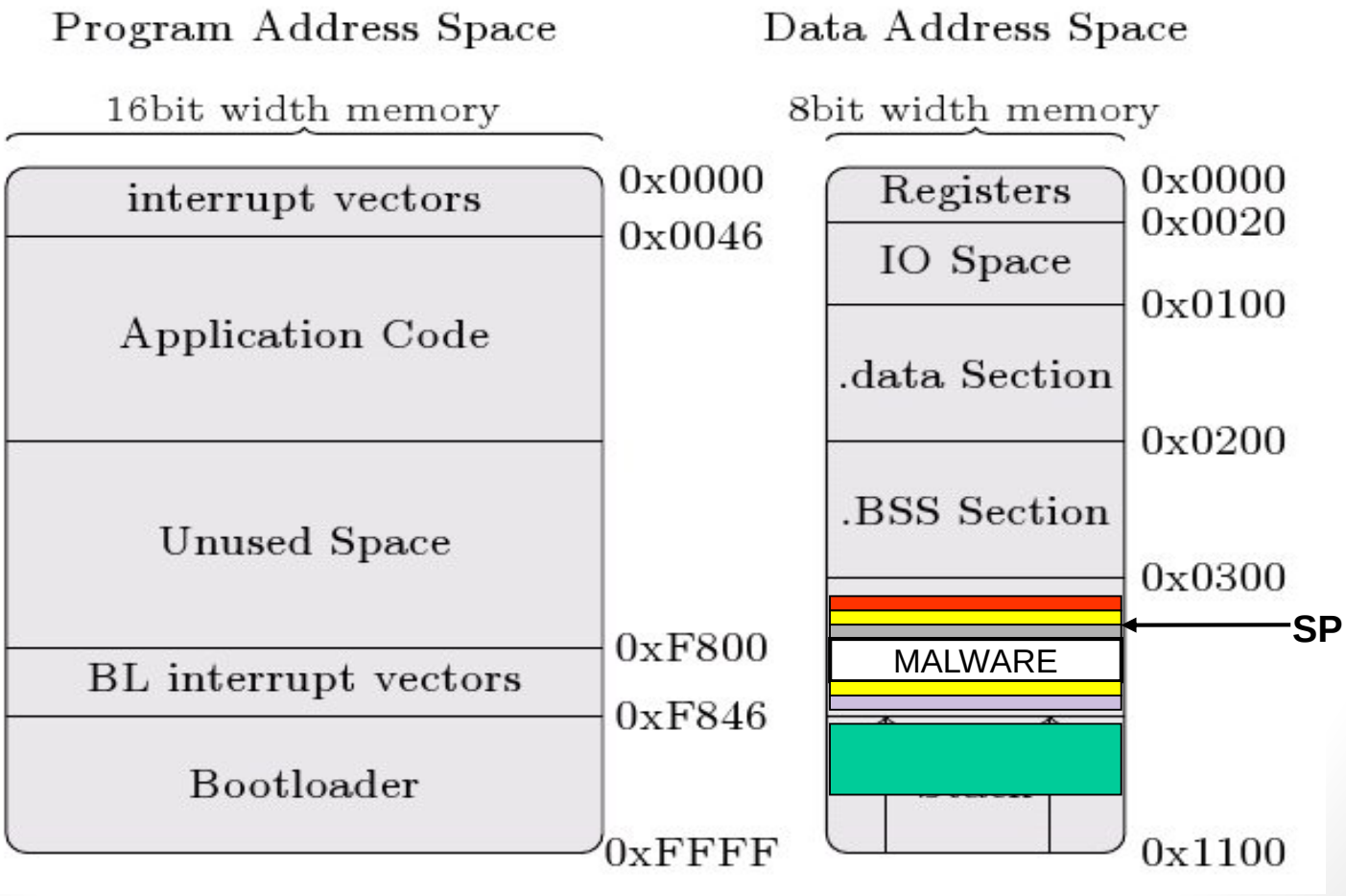
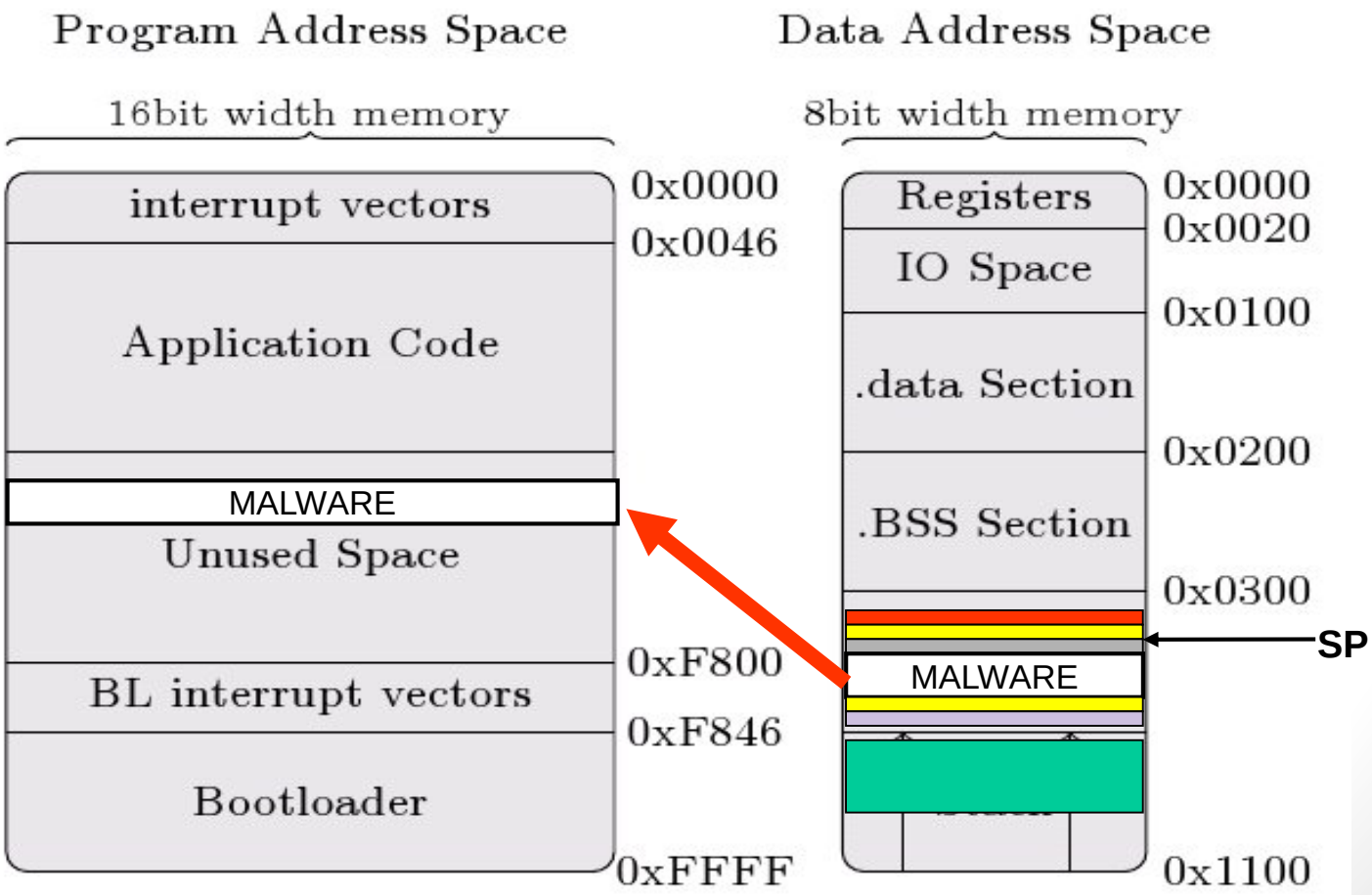


Illustration (9)

- The malware is copied from RAM to FLASH



Summary

- Harvard Architecture does not prevent code injection
- Return Oriented Programming can be used on constrained platforms
  - For a limited set of “actions”
- ROP can be used to inject code on AVR cpu
  - With small program size
  - Validated on 10 TinyOS applications
- The attack can be automated
  - tool that builds data injection payload automatically designed
- Worms and Viruses are realistic threats to Wireless sensor networks