Countering Trusting Trust through Diverse Double-Compiling

David A. Wheeler

February 28, 2006

http://www.dwheeler.com/trusting-trust

This presentation contains the views of the author and does not necessarily indicate endorsement by IDA, the U.S. government, or the U.S. DoD.

Outline

- Trusting trust attack
  - What it is
  - Attacker motivations
  - Triggers & payloads
- Inadequate solutions & related work
- Solution: Diverse double-compiling (DDC)
  - What it is
  - Why it works (assumptions, justification)
  - How to increase diversity
  - Practical challenges
- Demo: tcc
- Limitations & broader implications

Trusting trust attack

Compiler

malicious

executable

Critical program

source code

Compiler

malicious

executable

source code

Analysis program

source code

Trusted source...

malicious binaries

1974: Karger & Schell
1984: Ken Thompson. Demo'd (inc. disassembler), undetected

Fundamental security problem

Attacker motivations

- Huge benefits – Controlling a compiler controls everything it compiles
- Controlling 2-3 compilers would control almost every computer worldwide
- Risks low – no viable detection technique
- Costs low...medium
  - Requires one-time write of trusted binary
    - Not necessarily easy, but someone can, one-time, & not designed to withstand determined attack
  - Even if costs were high, the power to control every computer would be worth it to some

Triggers & payloads

- Attack depends on triggers & payloads
  - Trigger: code detects condition for performing malicious event (in compilation)
  - Payload: code performs malicious event (i.e., inserts malicious code)
- Triggers or payloads can fail
  - Change in source can disable trigger/payload
- Attackers can easily counter
  - Insert multiple attacks, each narrowly scoped
  - Refresh periodically via existing compromises

Inadequate solutions & Related work

- Manual binary review: Size, subverted tools
- Automated review / proof of binaries: Hard
- Recompile compiler yourself: Fails if orig. compiler malicious, massive diligence
- Interpreters just move attack location
- Draper/McDermott: Compile paraphrased source or with 2nd compiler, then recompile
  - Any who care must recompile their compilers
  - Can’t accumulate trust – can still get subverted
  - Helps; another way to use 2nd compiler?
Solution: Diverse double-compiling

- Developed by Henry Spencer in 1998
  - Check if compiler can self-regenerate
  - Compile source code twice: once with a second “trusted” compiler, then again using result
  - If result bit-for-bit identical to original, then source and binary correspond
- Never described/examined/justified in detail
- Never tried

Why does it work?

Assuming:
1. Have trusted: compiler T, DDC environment, comparer, process to get s_A & A
   - Trusted = triggers/payloads, if any, are different
2. T has same semantics as A for what’s in s_A
3. Flags etc. affecting output identical
4. Compiler s_A deterministic (control seed if random)

Then:
1. c(s_A,T) functionally same as A – same source code!
2. If A malicious, doesn’t matter – never run in DDC!
3. Final result bit-for-bit equal iff s_A represents A – only an untainted compiler, with identical functionality, creates the final result!

How to increase diversity

- Trusted Compiler T must not have triggers/payloads for compiler A
- Could prove T’s binary – hard
- Alternative: increase diversity
  - Compiler implementation (maximally different)
  - Time (esp. old compiler as trusted compiler)
  - Environment
  - Source code mutation/paraphrasing

Practical challenges

- Uncontrolled nondeterminism
  - May be no alternative compiler that can handle s
  - Can create, or hard-preprocess
  - “Pop-up” attack
    - Attacker includes self-perpetuating attack in only some versions (once succeeds, it disappears)
    - Defenders must thoroughly examine every version they accept, not just begin/end points
- Multiple compiler components
- Malicious environment? Redefine A as OS
- Inexact comparison (e.g., date/time stamp)

Demo: tcc

- Performed on small C compiler, tcc
  - Separate runtime library, handle in pieces
- tcc defect: fails to sign-extend 8-bit casts
  - x86: Constants -127..128 can be 1 byte (vs. 4)
  - tcc detects this with a cast (prefers short form)
  - tcc bug – cast produces wrong result, so tcc compiled-by-self always uses long form
- tcc junk bytes: long double constant
  - Long double uses 16 bytes, stored in 12 bytes
- Other two “junk” bytes have random data
- Fixed tcc, technique successfully verified fixed tcc
- Used verified fixed tcc to verify original tcc

It works!
Diverse double-compilation of tcc

Limitations

- Not absolute proof (unless T & environment proved)
- Hard to overcome & can use more tests/diversity
- Only shows source & binary correspond
- Could still have malicious code in source
- But we have techniques to address that!
- A’s source code must be available (easier for FLOSS)
- Source binary correspondence primarily useful if you can see compiler source
- Not yet demonstrated on larger scale – doing that now
- Easier if language standard & no software patents
  Visual Basic patent app for “isNot” operator

Broader implications

- Practical counter for trusting trust attack
- Can expand to TCB, whole OS, & prob. hardware
- Governments could require info for evals
- Receive all source code, inc. build instructions:
  - Of compilers: so can check them this way
  - Of non-compilers: check by recompiling
- Could establish groups to check major compiler releases for subversion
- Insist languages have public unpatented specifications (anyone can implement, any license)
- Source code examination now justifiable

Recent Work: Relaxing Constraint: Compiler Need not be Self-compiled

- Instead of self-compiling, can use parent compiler P
- P may be just a different version of A
- Source code s is now s_A union s_P
- Needs examining
- Can be used to “break” a loop

Backup

- Published Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC), December 2005, “Countering Trusting Trust through Diverse Double-Compiling”
- Required reading: Northern Kentucky University’s CSC 593: Secure Software Engineering Seminar, Spring 06
- Bruce Schneier’s weblog and Crypto-Gram
Can DDC be used with hardware?

- Probably; not as easy for pure hardware
- Requires 2nd implementation T
  - Alternative hardware compiler, simulated chip
- Requires “equality” test
  - Scanning electron microscope, focused ion beam
- Requires knowing exact correct result
  - Often cell libraries provided to engineer are not the same as what is used in the chip
  - Quantum effect error corrections for very high densities considered proprietary by correctors
- Only shows the chip-under-test is good

Can this scale up?

- Believe so; best proved by demonstration
- Working with “real” compiler: gcc
- Step 1: Real compiler, less diversity
  - A = Fedora Core 4’s gcc4
  - T/Environment = gcc3/Fedora Core 3
- Clarifies process, identifies dependencies
- Step 2: Real compiler, massive diversity
  - A = Fedora Core 4’s gcc4
  - T/Environment = SGI IRIX
- May change as learn more
  - Big challenge: Vendors don’t store info

Threat: Trusting trust attack

- First publicly noted by Karger & Schell, 1974
- Publicized by Ken Thompson, 1984
- Back door in “login” source code would be obvious
- Could insert back door in compiler source; login’s source is clean, compiler source code is not
- Modify compiler to also detect itself, and insert those attacks into compilers’ binary code
- Source code for login and compiler pristine, yet attack perpetuates even when compiler modified
- Can subvert analysis tools too (e.g., disassembler)
- Thompson performed experiment - never detected

Fundamental security problem
Countering Trusting Trust through Diverse Double-Compiling

David A. Wheeler

February 28, 2006
(Minor update from December 2, 2005)

http://www.dwheeler.com/trusting-trust

This presentation contains the views of the author and does not necessarily indicate endorsement by IDA, the U.S. government, or the U.S. DoD.
Outline

• Trusting trust attack
  – What it is
  – Attacker motivations
  – Triggers & payloads

• Inadequate solutions & related work

• Solution: Diverse double-compiling (DDC)
  – What it is
  – Why it works (assumptions, justification)
  – How to increase diversity
  – Practical challenges

• Demo: tcc

• Limitations & broader implications
Trusting trust attack

Trustworthy source…

- Critical program source code “login”
- Analysis program source code
- Compiler source code

… malicious binaries

- Critical program (malicious)
- Analysis program (malicious)
- Compiler executable (malicious)

1974: Karger & Schell
1984: Ken Thompson. Demo’d (inc. disassembler), undetected

Fundamental security problem
Attacker motivations

- Huge benefits – Controlling a compiler controls everything it compiles
  - Controlling 2-3 compilers would control almost every computer worldwide
- Risks low – no viable detection technique
- Costs low...medium
  - Requires one-time write of trusted binary
    - Not necessarily easy, but someone can, one-time, & not designed to withstand determined attack
  - Even if costs were high, the power to control every computer would be worth it to some
Triggers & payloads

- Attack depends on triggers & payloads
  - Trigger: code detects condition for performing malicious event (in compilation)
  - Payload: code performs malicious event (i.e., inserts malicious code)
- Triggers or payloads can fail
  - Change in source can disable trigger/payload
- Attackers can easily counter
  - Insert multiple attacks, each narrowly scoped
  - Refresh periodically via existing compromises
Inadequate solutions & Related work

• Manual binary review: Size, subverted tools
• Automated review / proof of binaries: Hard
• Recompile compiler yourself: Fails if orig. compiler malicious, massive diligence
• Interpreters just move attack location
• Draper/McDermott: Compile paraphrased source or with 2nd compiler, then recompile
  – Any who care must recompile their compilers
  – Can't accumulate trust – can still get subverted
  – Helps; another way to use 2nd compiler?
Solution: Diverse double-compiling

- Developed by Henry Spencer in 1998
  - Check if compiler can self-regenerate
  - Compile source code twice: once with a second “trusted” compiler, then again using result
  - If result bit-for-bit identical to original, then source and binary correspond
- Never described/examined/justified in detail
- Never tried
Diverse double-compiling in pictures

Key
- Compiler X
- Source Code SC
- Other input
- Compilation Result c(SC, X)

Can A regenerate?
- Self-regenerate?
- Compare 1
  - c(s_A, A)
- Compare 2
  - c(s_A, c(s_A, T))

Does s_A represent A?
Why does it work?

Assuming:

1. Have trusted: compiler T, DDC environment, comparer, process to get $s_A$ & A
   
   Trusted = triggers/payloads, if any, are different
2. T has same semantics as A for what's in $s_A$
3. Flags etc. affecting output identical
4. Compiler $s_A$ deterministic (control seed if random)

Then:

1. $c(s_A,T)$ functionally same as A – same source code!
2. If A malicious, doesn't matter – never run in DDC!
3. Final result bit-for-bit equal iff $s_A$ represents A – only an untainted compiler, with identical functionality, creates the final result!
How to increase diversity

- Trusted Compiler T must not have triggers/payloads for compiler A
- Could prove T's binary – hard
- Alternative: increase diversity
  - Compiler implementation (maximally different)
  - Time (esp. old compiler as trusted compiler)
  - Environment
  - Source code mutation/paraphrasing
Practical challenges

• Uncontrolled nondeterminism
• May be no alternative compiler that can handle s
  – Can create, or hand-preprocess
• “Pop-up” attack
  – Attacker includes self-perpetuating attack in only some versions (once succeeds, it disappears)
  – Defenders must thoroughly examine every version they accept, not just begin/end points
• Multiple compiler components
• Malicious environment? Redefine A as OS
• Inexact comparison (e.g., date/time stamp)
Demo: tcc

- Performed on small C compiler, tcc
  - Separate runtime library, handle in pieces
- tcc defect: fails to sign-extend 8-bit casts
  - x86: Constants -127..128 can be 1 byte (vs. 4)
  - tcc detects this with a cast (prefers short form)
  - tcc bug – cast produces wrong result, so tcc compiled-by-self always uses long form
- tcc junk bytes: long double constant
  - Long double uses 10 bytes, stored in 12 bytes
  - Other two “junk” bytes have random data
- Fixed tcc, technique successfully verified fixed tcc
- Used verified fixed tcc to verify original tcc

*It works!*
Diverse double-compilation of tcc

(Runtime) [s_{libtcc1}, s_{tcc}]  

(Rest of compiler) 

Must handle real compilers in pieces; the approach works

Diverse Double-compile

Stage 1

Stage 2

Self-regen?
Limitations

- Not absolute proof (unless T & environment proved)
  - But you can make as strong as you wish
  - Hard to overcome & can use more tests/diversity
- Only shows source & binary correspond
  - Could still have malicious code in source
  - But we have techniques to address that!
- A's source code must be available (easier for FLOSS)
- Source/binary correspondence primarily useful if you can see compiler source
- Not yet demonstrated on larger scale – doing that now
- Easier if language standard & no software patents
  - Visual Basic patent app for “isNot” operator
Broader implications

• Practical counter for trusting trust attack
• Can expand to TCB, whole OS, & prob. hardware
• Governments could require info for evals
  – Receive all source code, inc. build instructions:
    • Of compilers: so can check them this way
    • Of non-compilers: check by recompiling
  – Could establish groups to check major compiler releases for subversion
• Insist languages have public unpatented specifications (anyone can implement, any license)
• Source code examination now justifiable
In the News...

- Published *Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*, December 2005, “Countering Trusting Trust through Diverse Double-Compiling”
- Required reading: Northern Kentucky University's CSC 593: Secure Software Engineering Seminar, Spring 06
- Referenced in Bugtraq, comp.risks (Neumann's Risks digest), Lambda the ultimate, SC-L (the Secure Coding mailing list), LinuxSecurity.com, Chi Publishing's Information Security Bulletin, Wikipedia ("Backdoor"), Open Web Application Security Project (OWASP)
- Bruce Schneier's weblog and Crypto-Gram
Recent Work: Relaxing Constraint: Compiler Need not be Self-compiled

- Instead of self-compiling, can use parent compiler P
  - P may be just a different version of A

- Source code s is now \( s_A \cup s_P \)
  - Needs examining
  - If similar, diff

- Can be used to "break" a loop

\[ A_0 = c(s_A, P) \]
\[ P_1 = c(s_P, T) \]
\[ A_2 = c(s_A, c(s_P, T)) \]
Backup
Can DDC be used with hardware?

• Probably; not as easy for pure hardware
• Requires 2\textsuperscript{nd} implementation $T$
  - Alternative hardware compiler, simulated chip
• Requires “equality” test
  - Scanning electron microscope, focused ion beam
• Requires knowing exact correct result
  - Often cell libraries provided to engineer are \textit{not} the same as what is used in the chip
  - Quantum effect error corrections for very high densities considered proprietary by correctors
• Only shows the chip-under-test is good
Can this scale up?

• Believe so; best proved by demonstration
• Working with “real” compiler: gcc
• Step 1: Real compiler, less diversity
  – $A = \text{Fedora Core 4’s gcc4}$
  – $T/\text{Environment} = \text{gcc3/Fedora Core 3}$
  – Clarifies process, identifies dependencies
• Step 2: Real compiler, massive diversity
  – $A = \text{Fedora Core 4’s gcc4}$
  – $T/\text{Environment} = \text{SGI IRIX}$
• May change as learn more
  – Big challenge: Vendors don't store info
Threat: Trusting trust attack

- First publicly noted by Karger & Schell, 1974
- Publicized by Ken Thompson, 1984
  - Back door in “login” source code would be obvious
  - Could insert back door in compiler source; login's source is clean, compiler source code is not
  - Modify compiler to also detect itself, and insert those attacks into compilers' binary code
  - Source code for login and compiler pristine, yet attack perpetuates even when compiler modified
  - Can subvert analysis tools too (e.g., disassembler)
  - Thompson performed experiment - never detected

**Fundamental security problem**